Program Analysis and Specialization for the C Programming Language

Ph.D. Thesis

Lars Ole Andersen

DIKU, University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen Ø
Denmark
email: `lars@diku.dk`

May 1994

Abstract

Software engineers are faced with a dilemma. They want to write general and well-structured programs that are flexible and easy to maintain. On the other hand, generality has a price: *efficiency*. A specialized program solving a particular problem is often significantly faster than a general program. However, the development of specialized software is time-consuming, and is likely to exceed the production of today's programmers. New techniques are required to solve this so-called software crisis.

Partial evaluation is a program specialization technique that reconciles the benefits of generality with efficiency. This thesis presents an automatic *partial evaluator* for the Ansi C programming language.

The content of this thesis is *analysis* and *transformation* of C programs. We develop several analyses that support the transformation of a program into its generating extension. A generating extension is a program that produces specialized programs when executed on parts of the input.

The thesis contains the following main results.

- We develop a *generating-extension transformation*, and describe specialization of the various parts of C, including pointers and structures.
- We develop constraint-based inter-procedural *pointer* and *binding-time analysis*. Both analyses are specified via non-standard type inference systems, and implemented by constraint solving.
- We develop a *side-effect* and an *in-use analysis*. These analyses are developed in the classical monotone data-flow analysis framework. Some intriguing similarities with constraint-based analysis are observed.
- We investigate *separate* and *incremental* program analysis and transformation. Realistic programs are structured into modules, which break down inter-procedural analyses that need global information about functions.
- We prove that partial evaluation at most can accomplish *linear speedup*, and develop an automatic *speedup analysis*.
- We study the stronger transformation technique *driving*, and initiate the development of *generating super-extensions*.

The developments in this thesis are supported by an implementation. Throughout the chapters we present empirical results.

Preface

This thesis is submitted in fulfillment of the requirements for a Ph.D. degree in Computer Science at DIKU, the department of Computer Science, University of Copenhagen. It reports work done between February 1992 and May 1994. Supervisor was Prof. Neil D. Jones, DIKU, and external examiner was Dr. Peter Lee, Carnegie Mellon University.

The thesis consists of eleven chapters where the first serves as an introduction, and the last holds the conclusion. Several of the chapters are based on published papers, but they have either been extensively revised or completely rewritten. The individual chapters are almost self-contained, but Chapter 2 introduces notations used extensively in the rest of the thesis.

An overview of the thesis can be found as Section 1.5.

Acknowledgements

I am grateful to my advisor Neil D. Jones for everything he has done for me and my academic career. Without Neil I would probably not have written this thesis, and he has always provided me with inspirations, comments, and insight — sometimes even without being aware of it. Further, Neil has founded the TOPPS programming language group at DIKU, and continues to enable contacts to many great people around the world. Without the TOPPS group, DIKU would not be a place to be.

I would like to thank Peter Lee for his interest in the work, and for useful feedback, many comments and discussions during his stay at DIKU.

Special thanks are due to Peter Holst Andersen. He has made several of the experimental results reproduced in this thesis, and spent many hours on correcting (embarrassing) bugs in my code.

Olivier Danvy deserves thanks for his enthusiasm and the very useful feedback I received while wrote the chapter to the Partial Evaluation book. His comments certainly improved the book chapter, and have also influenced the presentation in this thesis, —- and my view of computer science.

I would like to thank Robert Glück for his continued interest in this work, and for many useful discussions about partial evaluation and optimization. Furthermore, I would like to thank Carsten Gomard; the paper on speedup analysis was written together with him.

Warm thanks go to the members of the TOPPS programming language group and its visitors for always providing a stimulating, enjoyable and also serious research environment. Its many active members are always ready to provide comments, share ideas and interests, and have created several contacts to researcher at other universities. The many “chokoladeboller”-mornings have clearly demonstrated that the TOPPS group is much more than just a collection of individual persons.

From the undergraduate and graduate days I would especially like to thank Jens Markussen. It was always fun to make written project, and later has many beers reminded me that there also is a world outside TOPPS and DIKU.

Lars K. Lassen and Berit Søemosegaard have, luckily!, dragged me away from boring work an uncountable number of times. It is always nice when they come by and suggest a cup of coffee, a beer, a game of billiard . . .

Finally, I would like to thank my parents and the rest of my family for everything they have done for me.

This research was supported by grant 16-5031 (“forskningsstipendium”) from the Danish Research Council STVF, and by the SNF research project DART, the EC ESPRIT BRA “Semantique” and DIKU.

Contents

Preface ii

Acknowledgements ii

1 Introduction 1
- 1.1 Software engineering 1
  - 1.1.1 From specification to deliverable 1
  - 1.1.2 The problem of excess generality versus efficiency 2
  - 1.1.3 Executable specifications 3
  - 1.1.4 Generality in programs 3
  - 1.1.5 Computation in stages 5
  - 1.1.6 Specialization applied to software development 6
- 1.2 Program specialization and partial evaluation 7
  - 1.2.1 Programs, semantics and representations 7
  - 1.2.2 The Futamura projections 7
  - 1.2.3 Generating extensions 9
- 1.3 Program analysis 9
  - 1.3.1 Approximation and safety 10
  - 1.3.2 Type-based analysis specifications 10
  - 1.3.3 Program analysis techniques 11
- 1.4 Contributions 12
  - 1.4.1 Overview of C-Mix 13
  - 1.4.2 Main results 13
  - 1.4.3 How this work differs from other work 14
  - 1.4.4 Why C? 15
- 1.5 Overview of thesis 16

2 The C Programming Language 17
- 2.1 Introduction 18
  - 2.1.1 Design motivations 18
  - 2.1.2 Conforming and strictly-conforming programs 19
  - 2.1.3 User feedback 20
  - 2.1.4 Program representation 20
  - 2.1.5 Notation and terminology 21
  - 2.1.6 Overview of the chapter 21

2.2 The syntax and static semantics of C . . . 21
2.2.1 Syntax . . . 21
2.2.2 Static semantics . . . 23
2.2.3 From C to abstract C . . . 25
2.3 A representation of C programs . . . 25
2.3.1 Program representation . . . 26
2.3.2 Static-call graph . . . 27
2.3.3 Separation of types . . . 29
2.4 Dynamic semantics . . . 30
2.4.1 Notations and conventions . . . 31
2.4.2 Semantics of programs . . . 31
2.4.3 Semantics of functions . . . 32
2.4.4 Semantics of statements . . . 32
2.4.5 Semantics of expressions . . . 35
2.4.6 Semantics of declarations . . . 36
2.4.7 Memory management . . . 37
2.4.8 Some requirements . . . 37
2.5 Related work . . . 38
2.6 Summary . . . 38

3 Generating extensions **39**
3.1 Introduction . . . 40
3.1.1 The Mix-equation revisited . . . 41
3.1.2 Generating extensions and specialization . . . 42
3.1.3 Previous work . . . 44
3.1.4 Overview of chapter . . . 45
3.2 Case study: A generating string-matcher . . . 46
3.2.1 The string-matcher `strstr()` . . . 46
3.2.2 Binding time separation . . . 47
3.2.3 The generating extension `strstr-gen()` . . . 48
3.2.4 Specializing ‘`strstr()`’ . . . 52
3.3 Types, operators, and expressions . . . 52
3.3.1 Constants . . . 53
3.3.2 Data types and declarations . . . 53
3.3.3 Operators . . . 54
3.3.4 Type conversions . . . 54
3.3.5 Assignment operators . . . 55
3.3.6 Conditional expressions . . . 56
3.3.7 Precedence and order of evaluation . . . 56
3.3.8 Expression statements . . . 56
3.4 Control flow . . . 57
3.4.1 The pending loop . . . 57
3.4.2 If-else . . . 58
3.4.3 Switch . . . 60

3.4.4 Loops — while, do and for . . . 61
3.4.5 Goto and labels . . . 62
3.5 Functions and program structure . . . 63
3.5.1 Basics of function specialization . . . 63
3.5.2 Functions and side-effects . . . 65
3.5.3 Recursion and unfolding . . . 66
3.5.4 External variables . . . 67
3.5.5 Static variables . . . 68
3.5.6 Register variable . . . 68
3.5.7 Initialization . . . 68
3.6 Pointers and arrays . . . 68
3.6.1 Pointers and addresses . . . 68
3.6.2 Pointers and function arguments . . . 69
3.6.3 Pointers and arrays . . . 70
3.6.4 Address arithmetic . . . 71
3.6.5 Character pointers and functions . . . 72
3.6.6 Pointer arrays, pointers to pointers, multi-dimensional arrays . . . 72
3.6.7 Pointers to functions . . . 73
3.7 Structures . . . 75
3.7.1 Basics of structures . . . 75
3.7.2 Structures and functions . . . 76
3.7.3 Pointers to structures . . . 78
3.7.4 Self-referential structures . . . 79
3.7.5 Runtime allocation of structures . . . 79
3.7.6 Replacing dynamic allocation by static allocation . . . 81
3.7.7 Unions . . . 82
3.7.8 Bit-fields . . . 83
3.8 Input and output . . . 83
3.8.1 Standard input and output . . . 83
3.8.2 Case study: formatted output — printf . . . 84
3.8.3 Variable-length argument lists . . . 85
3.8.4 File access . . . 85
3.8.5 Error handling . . . 86
3.8.6 Miscellaneous functions . . . 86
3.9 Correctness matters . . . 88
3.10 Memory management . . . 93
3.10.1 Basic requirements . . . 93
3.10.2 The storage model . . . 94
3.10.3 State descriptions . . . 95
3.10.4 Object copy functions . . . 96
3.10.5 The seenB4 predicate . . . 98
3.10.6 Improved sharing of code . . . 99
3.10.7 Heap-allocated memory . . . 100
3.11 Code generation . . . 100

3.11.1 Algebraic reductions 100
3.11.2 Deciding dynamic tests 101
3.12 Domain of re-use and sharing 101
3.12.1 Domain of specialization 102
3.12.2 Domain of re-use 102
3.12.3 Live variables and domain of re-use 103
3.13 Specialization, sharing and unfolding 104
3.13.1 Sharing and specialization 104
3.13.2 Unfolding strategy 105
3.14 Imperfect termination 106
3.15 Related work 106
3.15.1 Partial evaluation: the beginning 106
3.15.2 Partial evaluation for imperative languages 107
3.15.3 Generating extension generators 107
3.15.4 Other transformations for imperative languages 108
3.16 Future work and conclusion 108
3.16.1 Further work 108
3.16.2 Conclusion 109

4 Pointer Analysis **111**
4.1 Introduction 112
4.1.1 What makes C harder to analyze? 112
4.1.2 Points-to analysis 113
4.1.3 Set-based pointer analysis 114
4.1.4 Overview of the chapter 114
4.2 Pointer analysis: accuracy and efficiency 115
4.2.1 Flow-insensitive versus flow-sensitive analysis 115
4.2.2 Poor man's program-point analysis 116
4.2.3 Intra- and inter-procedural analysis 116
4.2.4 Use of inter-procedural information 117
4.2.5 May or must? 118
4.3 Pointer analysis of C 118
4.3.1 Structures and unions 118
4.3.2 Implementation-defined features 119
4.3.3 Dereferencing unknown pointers 120
4.3.4 Separate translation units 120
4.4 Safe pointer abstractions 121
4.4.1 Abstract locations 121
4.4.2 Pointer abstraction 123
4.4.3 Safe pointer abstraction 124
4.4.4 Pointer analysis specification 125
4.5 Intra-procedural pointer analysis 129
4.5.1 Pointer types and constraint systems 129
4.5.2 Constraint generation 131

4.5.3 Completeness and soundness . . . 136
4.6 Inter-procedural pointer analysis . . . 136
4.6.1 Separating function contexts . . . 137
4.6.2 Context separation via static-call graphs . . . 137
4.6.3 Constraints over variant vectors . . . 138
4.6.4 Inter-procedural constraint generation . . . 139
4.6.5 Improved naming convention . . . 140
4.7 Constraint solving . . . 140
4.7.1 Rewrite rules . . . 140
4.7.2 Minimal solutions . . . 142
4.8 Algorithm aspects . . . 143
4.8.1 Representation . . . 143
4.8.2 Iterative constraint solving . . . 143
4.8.3 Correctness . . . 144
4.8.4 Complexity . . . 145
4.9 Experiments . . . 145
4.10 Towards program-point pointer analysis . . . 146
4.10.1 Program point is sequence point . . . 146
4.10.2 Program-point constraint-based program analysis . . . 147
4.10.3 Why Heintze's set-based analysis fails . . . 147
4.11 Related work . . . 148
4.11.1 Alias analysis . . . 148
4.11.2 Points-to analysis . . . 150
4.11.3 Approximation of data structures . . . 151
4.12 Further work and conclusion . . . 151
4.12.1 Future work . . . 151
4.12.2 Conclusion . . . 152

5 Binding-Time Analysis 153
5.1 Introduction . . . 154
5.1.1 The use of binding times . . . 154
5.1.2 Efficient binding-time analysis . . . 155
5.1.3 Related work . . . 155
5.1.4 The present work . . . 156
5.1.5 Overview of the chapter . . . 156
5.2 Separating binding times . . . 157
5.2.1 Externally defined identifiers . . . 157
5.2.2 Pure functions . . . 158
5.2.3 Function specialization . . . 159
5.2.4 Unions and common initial members . . . 159
5.2.5 Pointers and side-effects . . . 159
5.2.6 Run-time memory allocation . . . 159
5.2.7 Implementation-defined behaviour . . . 160
5.2.8 Pointers: casts and arithmetic . . . 160

5.3 Specifying binding times 161
5.3.1 Binding-time types 161
5.3.2 Binding time classifications of objects 163
5.3.3 The lift relation 163
5.3.4 Divisions and type environment 164
5.3.5 Two-level binding-time annotation 165
5.3.6 Well-annotated definitions 165
5.3.7 Well-annotated expressions 167
5.3.8 Well-annotated statements 170
5.3.9 Well-annotated functions 171
5.3.10 Well-annotated C programs 172
5.4 Binding time inference 173
5.4.1 Constraints and constraint systems 173
5.4.2 Binding time attributes and annotations 174
5.4.3 Capturing binding times by constraints 175
5.4.4 Normal form 181
5.4.5 Solving constraints 182
5.4.6 Doing binding-time analysis 184
5.4.7 From division to well-annotated program 184
5.4.8 Extensions 185
5.5 Efficient constraint normalization algorithm 185
5.5.1 Representation 185
5.5.2 Normalization algorithm 187
5.5.3 Complexity 187
5.5.4 Correctness 188
5.5.5 Further improvements 188
5.6 Polyvariant binding-time analysis 188
5.6.1 Polyvariant constraint-based analysis 189
5.6.2 Polyvariance and generation extensions 190
5.7 Examples 190
5.8 Related work 191
5.8.1 BTA by abstract interpretation 191
5.8.2 BTA by type inference and constraint-solving 191
5.8.3 Polyvariant BTA 192
5.9 Further work 192
5.9.1 Constraint solving, tracing and error messages 192
5.9.2 The granularity of binding times 193
5.9.3 Struct variants 193
5.9.4 Analysis of heap allocated objects 193
5.10 Conclusion 193

6 Data-Flow Analysis **194**
6.1 Introduction 195
6.1.1 Data-flow analysis framework 195
6.1.2 Solutions methods 196
6.1.3 Inter-procedural program analysis 196
6.1.4 Procedure cloning 197
6.1.5 Taming pointers 198
6.1.6 Overview of chapter 198
6.2 Side-effect analysis 199
6.2.1 May side-effect 199
6.2.2 Side-effects and conditional side-effects 199
6.2.3 Using side-effect information 200
6.2.4 Control dependence 200
6.2.5 Conditional may side-effect analysis 201
6.2.6 Doing side-effect analysis 204
6.3 Use analysis 204
6.3.1 Objects in-use 204
6.3.2 In-use and liveness 205
6.3.3 Using in-use in generating extensions 206
6.3.4 In-use analysis functions 206
6.3.5 Doing in-use analysis 207
6.3.6 An enhancement 209
6.4 Related work 209
6.4.1 Side-effect analysis 209
6.4.2 Live-variable analysis 210
6.4.3 Procedure cloning and specialization 210
6.5 Conclusion and Future work 210
6.5.1 Further work 210
6.5.2 Conclusion 210

7 Separate Program Analysis and Specialization **211**
7.1 Introduction 212
7.1.1 Partial evaluation and modules 212
7.1.2 Modules and generating extensions 212
7.1.3 Pragmatics 213
7.1.4 Analysis in three steps 214
7.1.5 Separate specialization 214
7.1.6 Overview of chapter 214
7.2 The problem with modules 215
7.2.1 External identifiers 215
7.2.2 Exported data structures and functions 215
7.2.3 Pure external functions 216
7.3 Separate binding-time analysis 216
7.3.1 Constraint-based binding-time analysis revisited 217

7.3.2 Inter-modular binding-time information 218
7.3.3 Binding-time signatures 219
7.3.4 Doing inter-modular binding-time analysis 220
7.3.5 Using binding-times 222
7.3.6 Correctness of separate analysis 222
7.4 Incremental binding-time analysis 223
7.4.1 Why do incremental analysis? 223
7.4.2 The basic idea 223
7.4.3 The components of an incremental constraint solver 224
7.4.4 Correctness of incremental binding-time analysis 227
7.4.5 Doing incremental binding-time analysis 228
7.5 Separate specialization 229
7.5.1 Motivation 229
7.5.2 Conflicts between global transformations and modules 230
7.5.3 Towards separate specialization 230
7.5.4 A example: specializing library-functions 231
7.6 Separate and incremental data-flow analysis 231
7.6.1 Separate pointer analysis 231
7.7 Related work 232
7.8 Further work and conclusion 233
7.8.1 Future work 233
7.8.2 Conclusion 233

8 Speedup: Theory and Analysis 234
8.1 Introduction 235
8.1.1 Prospective speedups 235
8.1.2 Predicting speedups 236
8.1.3 A reservation 236
8.1.4 Overview of chapter 236
8.2 Partial evaluation and linear speedups 237
8.2.1 Measuring execution times 237
8.2.2 Linear speedup 238
8.2.3 Some examples 238
8.2.4 No super-linear speedup! 239
8.3 Predicting speedups 241
8.3.1 Safety of speedup intervals 242
8.3.2 Simple loops and relative speedup 242
8.3.3 Doing speedup analysis 243
8.3.4 Experiments 245
8.3.5 Limitations and improvements 246
8.3.6 Variations of speedup analysis 247
8.4 Predicting speedups in generating extensions 248
8.4.1 Accounting for static values 248
8.4.2 Speedup analysis in generating extensions 248

8.5 Related work 249
8.5.1 Speedup in partial evaluation 249
8.5.2 Speedup analysis versus complexity analysis 249
8.5.3 Properties of optimized programs 250
8.6 Future work 250
8.6.1 Costs of instructions 250
8.6.2 Estimation of code size 251
8.6.3 Unsafe optimizations and super-linear speedup 251
8.7 Conclusion 251

9 Partial Evaluation in Practice 253
9.1 C-Mix: a partial evaluator for C 254
9.1.1 Overview 254
9.1.2 C-Mix in practice 254
9.2 Speed versus size 254
9.2.1 Data-dependent algorithms 254
9.2.2 Case study: binary search 256
9.3 Specialization and optimization 260
9.3.1 Enabling and disabling optimizations 260
9.3.2 Order of optimization 261
9.3.3 Some observations 263
9.4 Experiments 263
9.4.1 Lexical analysis 263
9.4.2 Scientific computing 264
9.4.3 Ray tracing 264
9.4.4 Ecological modelling 265
9.5 Conclusion and further work 266
9.5.1 Future work 266
9.5.2 Summary 266

10 Improvements and Driving 267
10.1 Introduction 268
10.1.1 A catalogue of transformation techniques 268
10.1.2 An example of a genius transformation 269
10.1.3 Overview of chapter 271
10.2 Case study: generation of a KMP matcher 271
10.2.1 A naive pattern matcher 271
10.2.2 Positive context propagation 272
10.2.3 A KMP matcher 273
10.3 Towards generating super-extensions 275
10.3.1 Online decisions 275
10.3.2 Representation of unknown values 275
10.3.3 Theorem proving 276
10.3.4 Context propagation 278
10.3.5 Is this driving? 279

10.4 Related work 279
10.5 Future work and Conclusion 280
10.5.1 Further work 280
10.5.2 Conclusion 280

11 Conclusion **281**
11.1 Future work 281
11.1.1 Program specialization and transformation 281
11.1.2 Pointer analysis 282
11.1.3 Binding-time analysis 282
11.1.4 Data-Flow analysis 282
11.1.5 Separate analysis 282
11.1.6 Speedup analysis 282
11.1.7 Experiments 283
11.1.8 Driving 283
11.2 Final remarks 283

Bibliography **284**

Danish Summary **297**

Chapter 1

Introduction

The need for computer software is constantly increasing. As the prices of computers fall, new areas suitable for automatic data processing emerge. Today's bottleneck is not production and delivery of hardware, but the specification, implementation and debugging of programs. Tools aiming at automating the software development are needed to surmount the so-called *software crisis*. This thesis presents one such promising tool: an automatic *partial evaluator*, or *program specializer*, for the ANSI C programming language. In particular, we describe a generating extension transformation, and develop several program analyses to guide the transformation.

1.1 Software engineering

Despite the rapid development in computer technology, software engineering is still a handicraft for ingenious software engineers. Much effort is put into the development of prototypes, implementation of deliverables, and debugging of those, but the demand for programs is likely to exceed the capacity of the current software production. In this section we sketch the contemporary development of software and some of its deficiencies. Next we explain how automatic *program specialization* can reconcile the benefits of general solutions and efficient specialized programs, enabling fast computer software production and maintenance.

1.1.1 From specification to deliverable

The development of a computer system usually undertakes several phases:

$$\boxed{\text{specification}} \rightarrow \boxed{\text{prototyping}} \rightarrow \boxed{\text{implementation}} \rightarrow \boxed{\text{debugging}} \rightarrow \boxed{\text{deliverable}}$$

where some (or all) steps may be iterated.

From an initial specification of the problem, a prototype implementation is created. The objective is real-testing and requirement study of the system's potential performances and features. Often, prototypes are implemented in high-level languages such as Lisp that offer a rich library of auxiliary functions, and there programmers are less burdened with implementation of primitive data structures, *e.g.* lists.

When the prototyping stage has completed, the prototype is normally thrown away, and the system re-implemented in a language where *efficiency* is of major concern. Currently, many systems are implemented in the pervasive C programing language due to its availability on almost all platforms, its efficiency, and its support of low-level operations.

However, a vast amount of work is invested in debugging and run-in of deliverables — an effort that presumably already has been put in to the implementation of the prototype version. Due to the more primitive basic operations — consider for example the differences between list manipulation in Lisp and C — errors not present in the prototype are likely to creep in, and debugging is an order magnitude harder (and more time consuming).

Ideally, an implementation should be rigorously showed correct before delivered, but in practice this is an impossible task in the case of continually changing programs. Instead, the project manager may estimate that "less than 1 % errors remain" and haughtily state it "correct".

Obviously, it would be preferable to base final implementations on the prototype system instead of having to start from scratch: many errors introduced in the final implementation have probably been fixed in the prototype version; prototypes are more amenable for changes and allow code re-use, and prototypes are often more cleanly structured and hence easier to maintain.

There is, however, one unquestionable reason to why the prototype's generality is not kept: *efficiency.*

1.1.2 The problem of excess generality versus efficiency

An important property of a prototype is the possibility to modify the system to accommodate new requirements and specifications. To fulfill this, prototypes are *parameterized* over specifications.

For example, in a data base program, the number of lines of output can be given as a specification parameter instead of being "hard-coded" as a constant into the program wherever used. If a customer wants to experiment with different amounts of output before deciding on the right number, this can easily be done. Had the number been specified several places in the system, insight and work-hours would be needed to change it.[1] Unfortunately, the passing around and testing of specification constants slows down program execution, and is therefore normally removed in a final implementation.

From the software developer's point of view, however, it would be desirable to make the parameterization a permanent part of the system. As all software engineers know, users all too often reverse their decisions and want the systems to be changed. A significant part of the money spent on software development is canalized into software maintenance.

With a few exceptions — see below — *speed* is almost always weighted over *generality.* Unacceptably long response times may be the result of superfluous generality, and faster computers are not the answer: the complexity of problems grows faster, with hitherto undreamt subjects qualified for parameterization, than the hardware technology.

This is why the use of *executable specifications* is not dominant in professional software engineering.

[1]Admittedly, this is a contrived example, but it illustrates the principles.

```
/* printf: output values according to format */
int printf(char *format, char **values)
{
   int n = 0;
   for (; *format != '\0'; format++)
      if (*format != '%') putc(*format);
      else switch (*++format) {
         case '%': putc('%'); break;
         case 's': puts(values[n++]); break;
         default : return EOF;
      }
   return n;
 }
```

Figure 1: Mini-version of the C standard-library function `printf()`

1.1.3 Executable specifications

A compelling idea is automatic transformation of specifications into efficient, specialized programs, the objective being faster software development and reduced software maintaining costs. For this to be possible, the specification must *executable* in some sense. In this thesis we are not concerned with transformation of very high level specifications to low-level code, but the optimization of target code produced by such transformations. For now, we use Lisp[2] as a specification language. We believe that most programmers will agree that software development in Lisp is more productive and less error-prone than in *e.g.* assembly code.

Automatic compilation of Lisp to C, say, does not solve the efficiency problem, though. Traditional compilation techniques are not strong enough to eliminate excessive generality. Using today's methods, the translated program will carry around the specification and perform testing on it at runtime. Furthermore, automatic transformation of high-level languages into low-level languages tends to employ very general compilation schemes for both statements and data structures. For example, instead of using separate, statically allocated variables, automatically produced programs use heap-allocated objects. Again we identify *excessive generality* as the obstacle for efficiency.

In conclusion: even though more efficient than the corresponding Lisp program, the transformed program may still be an order of magnitude slower than a "hand-made" piece of software. What is needed is a way of transforming a *general* program into a *specialized* program that eliminates specification testing.

1.1.4 Generality in programs

Excessive generality is not, however, inseparably coupled with automatic program transformation. It is, actually, present in most existing programs. Consider Figure 1 that depicts a mini-version of the C library function '`printf()`'.

The body of the function contains a loop that traverses through the '`format`' string.

[2]Substitute with a favorite programming language: SML, Ada, Haskell, ...

```
/* printf_spec: print "n=values[0]" */
int printf_spec(char **values)
{
   putc('n');
   putc('=');
   puts(values[0]);
   return 1;
}
```

Figure 2: Specialized version of '`printf()`*'*

According to the control codes in the string, different actions are performed. The time spent on determining *what* to do, *e.g.* to output '%', is normally referred to as the *interpretation overhead.*

The '`printf()`' function is a general one: it can produce all kinds of output according to the *specification* '`format`'. This makes the function a prime candidate for inclusion in a library, and for code re-use. The excess generality is seldom needed, though.

In a typical program, all calls to '`printf()` are of the form '`printf("n=%s", v)`', where the format is a constant string. Thus, the program fully specifies the action of '`printf()`' at compile-time; the interpretation overhead is wasted. The general call could profitably be replaced by a call to a *specialized* version of '`printf()`', as illustrated in Figure 2. In the specialized version, the specification has been "coded" into the control structure. The net effect is that the interpretation overhead has gone away.

The example generalizes easily: a differential equation solver is invoked several times on the same equation system with different start conditions; a program examining DNA-strings looks for the same pattern in thousands of samples; a ray-tracer is repeatedly applied to the same picture but with different view points; a spread-sheet program is run on the same tax base specification until the taxes change; an interpreter is executed several times on the same subject program but with different input.

The latter is a restatement of the well-known fact that compilation and running of a low-level target program are normally faster than interpretation of a high-level program. The reason is analogous to the other examples: the interpreter performs a great amount of run-time book-keeping that can be removed by compilation. Thus many existing program may also benefit form *specialization.*

More broadly, purely from efficiency criteria, specialization may be advantageous whenever one or more parameters change more frequently than others. As an example, if a program contains two calls with the same format string, the interpretation overhead is executed twice in the general version, and not at all in the specialized version.

On the other hand, only a few programmers actually write a specialized version of '`printf()`'; they use the general version. The reason is obvious: the existing version of '`printf()`' is known to be correct and the use of it does not introduce bugs, and it is convenient.

The analogy to software development via general prototypes should be clear. From a single prototype our wish is to derive several, specialized and efficient implementations. The prototype can then be included in a library and used repeatedly.

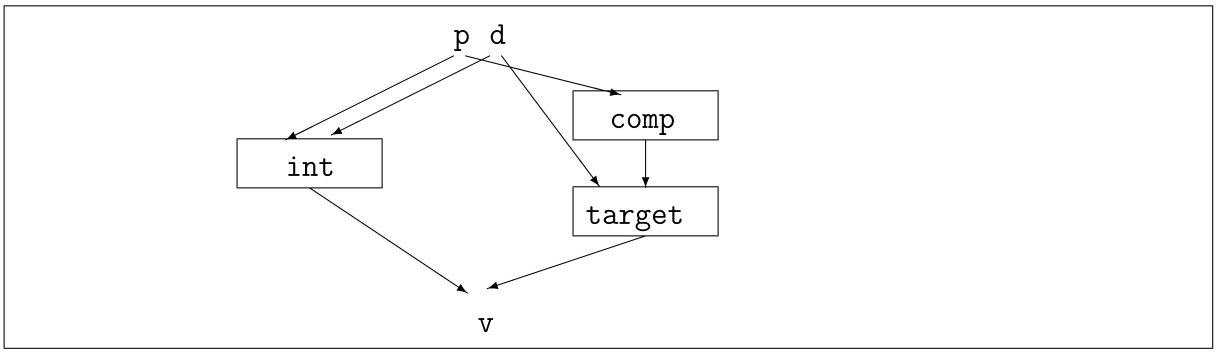

Figure 3: Computation in one or two stages

1.1.5 Computation in stages

Most programs contain computations which can be separated into *stages*; a notable example being interpreters.

An S-interpreter is a program *int* (in a language L) that takes as input a subject S-program *p* and the input *d* to *p*, and returns the same result as execution of *p* applied *d* on an S-machine. The notion is made precise in Section 1.2.1 below. An S-to-L compiler *comp* is a program that as input takes *p* and yields a *target* program p' in the L language. When run, the target program gives the same result as *p* applied to *d* on an S machine.

The two ways of execution a program is illustrated in Figure 3. To the left, the result *v* is produced in one step; to the right, two stages are used.

The computations carried out by a compiler are called *compile-time*, the other *run-time*. The interpreter *mixes* the times: during a run, the interpreter both performs compile-time computations (*e.g.* syntax analysis) and run-time calculations (*e.g.* multiplication of integers). In the '`printf()`' function, the compile-time computations include the scanning of the '`format`' string while the calls to '`putc()`' and '`puts()`' constitute the run-time part. Separation of the stages enables a performance gain.

Why are we then interested in interpreters, or more broadly, general programs, when generality often causes a loss in performance? The are a number of persuading arguments:

- General programs are often *easier* to write than specialized programs.[3]
- General programs support *code re-use*. Programs do not have not to be modified every time the contexts change, only the input specification.
- General programs are often uniform in structure, and hence more manageable to maintain and show correct. Furthermore, only one program has to be maintained as opposed to several dedicated (specialized) versions.

These have been well-known facts in the computer science for many years, and in one area, syntax analysis, *generality* and *efficiency* have successfully been reconciled.

[3]One reason is that programmers can ignore efficiency aspects whereas efficiency is a major concern when writing a specialized program for a particular task.

Implementation of a parser on the basis of a specification, *e.g.* a BNF-grammar, is a tedious, time-consuming and trivial job. An alternative is to employ a *general* parser, for example Earley's algorithm, that takes as input both a grammar and a stream of tokens. However, as witnessed by practical experiments, this yields excessively slow parsers for commercial software. In order to produce efficient parsers from BNF specifications, parser generators such as Yacc have been developed. Yacc produced parsers are easy to generate, and they are known to adhere to the specification.

A much admired generalization has been around for years: compiler generators, that convert language specifications into compilers. A fruitful way of specifying a language is by the means of an *operational semantics*. When put into a machine-readable form, an operational semantics is an interpreter. The present thesis also contributes in this direction.

Our approach goes beyond that of Yacc, however. Where Yacc is dedicated to the problem of parser generation, our goal is automatic generation of *program generators*. For example, given Earley's algorithm, to generate a parser generator, and when given an interpreter, to generate a compiler.

1.1.6 Specialization applied to software development

Today's professional software development is to a large extent dominated by the software engineers. To solve a given problem, a specific program is constructed — often from scratch. Even though the *products* of software companies have undergone a revolution, only little progress in the actual software development has happened.

Consider for example the area of data base programs. Years ago, a specialized data base for each application would have been developed by hand. Now, a general data base program is customized to handle several different purposes. Without these highly general data base programs, the technology would not exhibit the worldwide use, as is the case today. The extra generality has been made acceptably due to larger computers,[4] but, some problems require fast implementations which rule out excess generality.

This principle can be extended to the software engineering process itself. We imagine that future software engineering will be based on highly parameterized general programs, that automatically are turned into efficient implementations by the means of *program specialization*. This gives the best of the two worlds:

- Only one general program has to be maintained and shown correct.
- A new specialized program complying to changed specifications can be constructed automatically. The correctness comes for free.
- Code re-use is urged; by simply changing the parameters of the general program, many unrelated problems may be solvable by the same program.
- The programmer can be less concerned with efficiency matters.

This thesis develops such a software tool: an automatic program specializer for the C programming language.

[4]and that many small companies prefer a slow (and cheap) data base than no data base at all.

1.2 Program specialization and partial evaluation

In order to conduct analysis and transformation of a program, a clear specification of a program's *semantics* is needed. In this section we define the *extensional* meaning of a program run. For the aim of analysis and transformation, this is normally insufficient; an *intensional* specification, assigning meaning to the various language constructs, is required.

1.2.1 Programs, semantics and representations

To simplify the presentation we regard (for now) a program's meaning to be an input-output function. If p is a program in some programming language C over data domain V, $[\![p]\!]_C : V^* \to V_\perp$ denotes its *meaning* function. Since programs may fail to terminate, the lifted domain $V_\perp$ is used. In the case of the C programming language, $[\![\cdot]\!]$ is a complying implementation of the Standard [ISO 1990]. For a program p and input d we write $[\![p]\!]_C(d) \Rightarrow v$ if execution of p on d yields value v. For example, if a is an array containing the string '`"Hello"`', $[\![\texttt{printf}]\!]_C(\texttt{"n=\%s"}, a) \Rightarrow \texttt{"n=Hello"}$ (where we consider the function to be a program).

An S-interpreter is a C program *int* that takes as input a *representation* of an S program and a *representation* of its input d, and yields the same result as the meaning function specifies:

$$[\![\mathit{int}]\!](\overline{p}^{pgm}, \overline{d}^{val}) \Rightarrow \overline{v}^{val} \text{ iff } [\![p]\!]_S(d) \Rightarrow v$$

where $\overline{d}^{val}$ denotes the representation of an S-input in a suitable data structure, and similarly for $\overline{p}^{pgm}$.

A compiler *comp* can be specified as follows:

$$[\![\mathit{comp}]\!]_C(\overline{p}^{pgm}) \Rightarrow \overline{\mathit{target}}^{pgm}, [\![\mathit{target}]\!]_C(d) \Rightarrow v \text{ iff } [\![p]\!]_S(d) \Rightarrow v$$

for all programs p and input d.

The equation specifies that the compiler must adhere to the language's standard, *e.g.* a C compiler shall produce target programs that complies to the Standard [ISO 1990]. Admittedly, this can be hard to achieve in the case of languages with an involved dynamic semantics. This is why a compiler sometimes is taken to be *the* standard, *e.g.* as in "the program is ANSI C in the sense that '`gcc -ansi -Wall`' gives no warnings or errors".[5]

1.2.2 The Futamura projections

Let p be a C program and s, d inputs to it. A *residual* version p_s of p with respect to s is a program such that $[\![p_s]\!]_C(d) \Rightarrow v$ iff $[\![p]\!]_C(s, d) \Rightarrow v$.

The program p_s is a *residual*, or *specialized* version of p with respect to the known input s. The input d is called the unknown input.

A *partial evaluator* '`mix`' is a program that specializes programs to known input. This is captured in the Mix equation [Jones *et al.* 1989,Jones *et al.* 1993].

[5]Stolen from an announcement of a parser generator that was claimed to produce 'strict ANSI C'.

Definition 1.1 *Let p be a C program and s, d input to it. A* partial evaluator `mix` *fulfills*

$$\llbracket \texttt{mix} \rrbracket_C(\overline{p}^{pgm}, \overline{s}^{val}) \Rightarrow \overline{p_s}^{pgm} \text{ s.t. } \llbracket p_s \rrbracket_C(d) \Rightarrow v$$

whenever $\llbracket p \rrbracket_C(s, d) \Rightarrow v$. □

Obviously, '`mix`' must operate with the notions of compile-time and run-time. Constructs that solely depend on known input are called *static*; constructs that may depend on unknown input are called *dynamic*. The classification of constructs into static or dynamic can either be done before the actual specialization (by a *binding-time analysis*) or during specialization. In the former case the system is called *off-line*; otherwise it is *on-line*. We only consider off-line specialization in this thesis, for reason that will become apparent.[6]

Suppose that '`Cmix`' is a partial evaluator for the C language. Then we have:

$$\llbracket \texttt{Cmix} \rrbracket(\overline{\texttt{printf}}^{pgm}, \overline{\texttt{"n=\%s"}}^{val}) \Rightarrow \overline{\texttt{printf\_spec}}^{pgm}$$

where the program '`printf_spec()`' was listed in the previous section.

A remark: for practical purposes we will weaken the definition and allow '`mix`' to loop. In the following we implicitly assume that '`mix`' terminates (often enough) — not a problem in practice.

If the partial evaluator '`mix`' is written in the same language as it accepts, it is possible to *self-apply* it. The effect of this is stated in the Futamura projections [Jones *et al.* 1993].

Theorem 1.1 *Let int be an S-interpreter and 'p' an S program. Then holds:*

$$\begin{array}{lcl}
\llbracket \texttt{mix} \rrbracket_C(\overline{int}^{pgm}, \overline{p}^{val}) & \Rightarrow & \overline{target}^{pgm} \\
\llbracket \texttt{mix} \rrbracket_C(\overline{\texttt{mix}}^{pgm}, \overline{\overline{int}^{pgm}}^{val}) & \Rightarrow & \overline{comp}^{pgm} \\
\llbracket \texttt{mix} \rrbracket_C(\overline{\texttt{mix}}^{pgm}, \overline{\overline{\texttt{mix}}^{pgm}}^{val}) & \Rightarrow & \overline{cogen}^{pgm}
\end{array}$$

where comp is a compiler, and cogen a compiler generator.

The projections are straightforward to verify via the Mix equation, and the definitions of target programs and compilers.

The compiler generator '`cogen`' is a program that, given an interpreter, yields a compiler:

$$\llbracket \texttt{cogen} \rrbracket_C(\overline{\overline{int}^{pgm}}^{val}) \Rightarrow \overline{comp}^{pgm},$$

cf. the Mix equation and the second Futamura projection. Suppose that we apply '`cogen`' to the "interpreter" '`printf()`':

$$\llbracket \texttt{cogen} \rrbracket_C(\overline{\overline{\texttt{printf}}^{pgm}}^{val}) \Rightarrow \overline{\texttt{printf\_gen}}^{pgm}$$

where the "result" function is called '`printf_gen()`'. When run on the static input `"n=%s"`, the *extension* will *generate* a specialized program '`printf_spec()`'.

[6]This statement is not completely true. We will also investigate a 'mix-line' strategy where some decisions are taken during specialization.

1.2.3 Generating extensions

A *generating extension* is a program generator: when executed it produces a program.

Definition 1.2 *Let p be a program and s, d input to it. A* generating extension *p-gen of p yields a program p_s when executed on s,*

$$[\![p\text{-}gen]\!]_C(s) \Rightarrow \overline{p_s}^{pgm}$$

such that p_s is a specialized *version of p with respect to s.* □

A *generating-extension generator* is a program that transforms programs into generating extensions.

Definition 1.3 *A* generating extension generator *'`gegen`' is a program such that for a program p:*

$$[\![\texttt{gegen}]\!]_C(\overline{p}^{pgm}) \Rightarrow \overline{p\text{-}gen}^{pgm}$$

where p-gen is a generating extension of p. □

The compiler generator '`cogen`' produced by double self-application of '`mix`' is also a generating-extension generator. However, notice the difference in signature:

$$[\![\texttt{cogen}]\!]_C(\overline{\overline{p}^{pgm}}^{val}) \Rightarrow \overline{p\text{-}gen}^{pgm}$$

Where '`gegen`' operates on a (representation of a) program, '`cogen`' inputs a "val"-encoding of the program. The reason is that '`cogen`' inherits the representation employed by the '`mix`' from which it was generated.

1.3 Program analysis

A program transformation must be *meaning preserving* such that the optimized program computes the same result as the original program. Transformations based on purely syntactical criteria are often too limited in practice; information about construct's behaviour and result are needed.

The aim of *program analysis* is to gather dynamic properties about programs before they actually are run on the computer. Typical properties of interest include: which objects may a pointer point to? (pointer analysis, Chapter 4); is the variable '`x`' bound at compile-time or run-time? (binding-time analysis, Chapter 5); how much efficiency is gained by specializing this program? (speedup analysis, Chapter 8); and is this variable used in this function? (in-use analysis, Chapter 6).

The information revealed by a program analysis can be employed to guide transformations and optimizations. A compiler may use live-variable information to decide ether to preserve a value in a register; a partial evaluator may use binding-time information to decide whether to evaluate or reduce an expression, and the output of an analysis may even be employed by other program analyses. For example, in languages with pointers, approximation of pointer usage is critical for other analyses. Lack of pointer information necessitates worst-case assumptions which are likely to degrade the accuracy to trivia.

1.3.1 Approximation and safety

A program analysis is normally applied before program input is available, and this obviously renders the analysis undecidable. In complexity theory, this is known as Rice's Theorem: all but trivial properties about a program are undecidable. In practice we desire the analysis to compute information valid for *all* possible inputs.

This means, for example, that both branches of an `if` statement must be taken into account. In classical data-flow analysis this is known as the *control-flow assumption*: all control-paths in the program are executable. Of course, a "clever" analysis may recognize that the then-branch in the following program fragment is never executed,

```
if (0) x = 2; else x = 4;
```

but this only improves the analysis' precision on *some* programs — not in general.

Even under the control-flow assumption some properties may be undecidable; one such example is alias analysis in the presence of multi-level pointers, see Chapter 4. This implies that an analysis inevitably must *approximate.* Sometimes it fails to detect a certain property, but it must always output a *safe* answer. The notion of *safety* is intimately connected with both the aim of the analysis, and the consumer of the inferred information. For example, in the case of live-variable analysis, if is *safe* to classify as live all variables the pointer '`p`' may refer to in an assignment '`*p = 2`'. However, in the case of a constant propagation analysis, it would be erroneous to regard the same objects as constant.[7]

When it has been decided *which* properties an analysis is supposed to approximate, it must (ideally) be proved correct with respect to the language's semantics.

1.3.2 Type-based analysis specifications

A semantics is normally solely concerned with a program's input-output behavior. As opposed to this, program analysis typically collects informations about a program's interior computation. An example: the set of abstract locations a pointer may assume. This necessitates an *instrumented semantics* that collects the desired information as well as characterizes the input-output function. For example, an instrumented, or collecting semantics may accumulate all the values a variable can get bound to. An analysis can then be seen as a decidable abstraction of the instrumented semantics.

In this thesis we specify analyses as *non-standard type systems* on the basis of an operational semantics for C. Many properties of programs can be seen as *types* of expressions (or statements, functions *etc.*). Consider for example constant propagation analysis. The aim of this analysis is to determine whether variables possess a constant value. This is precisely the case if the part of the store S representing a variable has a constant value.[8]

The (dynamic) operational semantics for a small language of constants, variables and binary plus can be given as inference rules as follows

[7]In the latter case 'must point to' information is needed.

[8]For simplicity we only find variables that have the same constant values throughout the whole program.

$$\vdash \langle c, S\rangle \Rightarrow \langle c, S\rangle \qquad \vdash \langle v, S\rangle \Rightarrow \langle S(v), S\rangle \qquad \frac{\vdash \langle e_1, S\rangle \Rightarrow \langle v_1, S_1\rangle \quad \vdash \langle e_2, S_1\rangle \Rightarrow \langle v_2, S_2\rangle}{\vdash \langle e_1 + e_2, S\rangle \Rightarrow \langle v_1 + v_2, S_2\rangle}$$

where S is a store. A constant evaluates to itself; the value of a variable is determined by the store, and the value of a binary plus is the sum of the subexpressions.

To perform constant propagation analysis, we abstract the value domain and use the domain $\perp \sqsubset n \sqsubset \top$ where $\perp$ denotes "not used", $n \in \mathbb{N}$ means "constant value n", and $\top$ denotes "non-constant".

$$\vdash \left\langle c, \tilde{S}\right\rangle \Rightarrow \left\langle c, \tilde{S}\right\rangle \qquad \vdash \left\langle v, \tilde{S}\right\rangle \Rightarrow \left\langle \tilde{S}(v), \tilde{S}\right\rangle \qquad \frac{\vdash \left\langle e_1, \tilde{S}\right\rangle \Rightarrow \left\langle v_1, \tilde{S}_1\right\rangle \quad \vdash \left\langle e_2, \tilde{S}_1\right\rangle \Rightarrow \left\langle v_2, \tilde{S}_2\right\rangle}{\vdash \left\langle e_1 + e_2, \tilde{S}\right\rangle \Rightarrow \left\langle v_1 \tilde{+} v_2, \tilde{S}_2\right\rangle}$$

where the operator $\tilde{+}$ is defined by $m\tilde{+}n = m + n$, $m\tilde{+}\top = \top$, $\top\tilde{+}n = \top$ (and so on).

The analysis can now be *specified* as follows. Given a fixed *constant propagation map* $\tilde{S}$ (with the interpretation that $\tilde{S}(v) = n$ when v is constant n), the map is *safe* if it fulfills the rules

$$\tilde{S} \vdash c : c \qquad \tilde{S} \vdash v : \tilde{S}(v) \qquad \frac{\tilde{S} \vdash e_1 : v_1 \quad \tilde{S} \vdash e_2 : v_2}{\tilde{S} \vdash e_1 + e_2 : v_1 \tilde{+} v_2}$$

for an expression e. We consider the property "being constant" as a *type*, and have defined the operator $\tilde{+}$ (as before) on types.

As apparant from this example, the type-based view of program analysis admits easy comparison of the language's dynamic semantics with the specifications, to check safety and correctness.

1.3.3 Program analysis techniques

Program analysis techniques can roughly be grouped as follows.

- Classical data-flow analysis, including *iterative* [Hecht and Ullman 1975] and *elimination* or *interval* [Ryder and Paull 1986] methods. See Marlow and Ryder for a survey [Marlowe and Ryder 1990b].

- Abstract interpretation, including *forward* [Cousot and Cousot 1977] and *backward* [Hughes 1998] analysis. See Jones and Nielson for a survey [Jones and Nielson 1994].

- Non-standard type inference, including "*algorithm W*" methods [Gomard 1990], and *constraint-based* techniques [Henglein 1991,Heintze 1992].

Classical data-flow analyses are based on data-flow equation solving. Iterative methods find a solution by propagation of approximative solutions until a fixed-point is reached. Interval methods reduce the equation system such that a solution can directly be computed. The data-flow equations are typically collected by a syntax-directed traversal over the program's syntax tree. For example, in live-variable analysis [Aho *et al.* 1986], gen/kill equations are produced for each assignments.

Classical data-flow analysis is often criticized for being ad-hoc and not semantically based. In the author's opinion this is simply because most classical data-flow analyses were formulated before the framework of both denotational and operational semantics were founded. Moreover, many classical data-flow analyses are aimed at languages such as Fortran with a rather obvious semantics, *e.g.* there are no pointers. Finally, it is often overlooked that large part of the literature is concerned with *algorithmic* aspects contrary to the literature on abstract interpretation, which is mainly concerned with specification.

Abstract interpretation is usually formulated in a denotational semantics style, and hence less suitable for imperative languages. A "recipe" can be described as follows: first, the standard semantics is instrumented to compute the desired information. Next the semantics is abstracted into a decidable but approximate semantics. The result is an executable specification.

In practice, abstract interpretations are often implemented by simple, naive fixed-point techniques, but clever algorithms have been developed. It is, however, common belief that abstract interpretation is too slow for many realistic languages. On reason is that the analyses traverse the program several times to compute a fixed-point. Ways to reduce the runtime of "naive" abstract interpretations have been proposed, borrowing techniques from classical data-flow analysis, *e.g.* efficient fixed-point algorithms.

An expression's type is an abstraction of the set of values it may evaluate to. As we have seen, many data-flow problems can been characterized as *properties* of expressions, and specified by the means of non-standard type systems. The non-standard types can then be *inferred* using standard techniques, or more efficient methods, *e.g.* constraint solving. Hence, a non-standard type-based analysis consists of two separated parts: a *specification*, and an *implementation.* This eases correctness matters since the specification is not concerned with *efficiency.* On the other hand, the type-based specification often gives a uniform formulation of the computational problem, and allows efficient algorithms to be developed. For example, the "point-to" problem solved in Chapter 4 is reduced into solving a system of inclusion constraints.

In this thesis the type-based analyses are dominating, and in particular constraint-based analyses. This gives a surprisingly simple framework. Furthermore, the relation between specification and implementation is clearly separated, as opposed to most classically formulated data-flow problems.

We shall also, however, consider some data-flow problems which we formulate in the framework of classical monotone data-flow analysis, but on a firm semantical foundation. As a side-effect we observe that classical data-flow analyses and constraint-based analyses have many similarities.

1.4 Contributions

This thesis continues the work on automatic analysis and specialization of realistic imperative languages, and in particular C. A main result is *C-Mix*, an automatic partial evaluator for the ANSI C programming language.

1.4.1 Overview of C-Mix

C-Mix is a partial evaluator for the ANSI C programming language. Its phases are illustrated in Figure 4.[9]

First, the subject program is parsed and an intermediate representation built. During this, type checking and annotation is carried out, and a call-graph table is built.

Next, the program is analyzed. A *pointer analysis* approximates the usage of pointers; a *side-effect analysis* employs the pointer information to detect possible side-effects; an *in-use analysis* aims at finding "useless" variables; a *binding-time analysis* computes the binding time of expressions; and a *speedup analysis* estimates the prospective speedup.

The *generating extension generator* transforms the binding-time annotated program into a generating extension. The generating extension is linked with a library that implements common routines to yield an executable.

The chapters of this thesis describe the phases in detail.

1.4.2 Main results

This thesis contributes with the following results.

- We develop and describe the transformation of a binding-time analyzed ANSI C program into its generating extension, including methods for specialization of programs featuring pointers, dynamic memory allocation, structs and function pointers.
- We develop an inter-procedural pointer analysis based on constraint solving.
- We develop a constraint-based binding-time analysis that runs in almost-linear time.
- We develop a side-effect and an in-use analysis, formulated as classical monotone data-flow problems.
- We investigate the potential speedup obtainably via partial evaluation, and devise an analysis to estimate the speedup obtainable from specialization.
- We develop a separate and incremental binding-time analysis, and describe separate program specialization. Moreover, we consider separate analysis more broadly.
- We study stronger transformations and specialization techniques than partial evaluation, the foremost being driving.
- We provide experimental result that demonstrates the usefulness of *C-Mix*.

An implementation of the transformations and analyses supports the thesis.

[9]It should be noted that at the time of writing we have not implemented or integrated all the analyses listed in the figure. See Chapter 9.

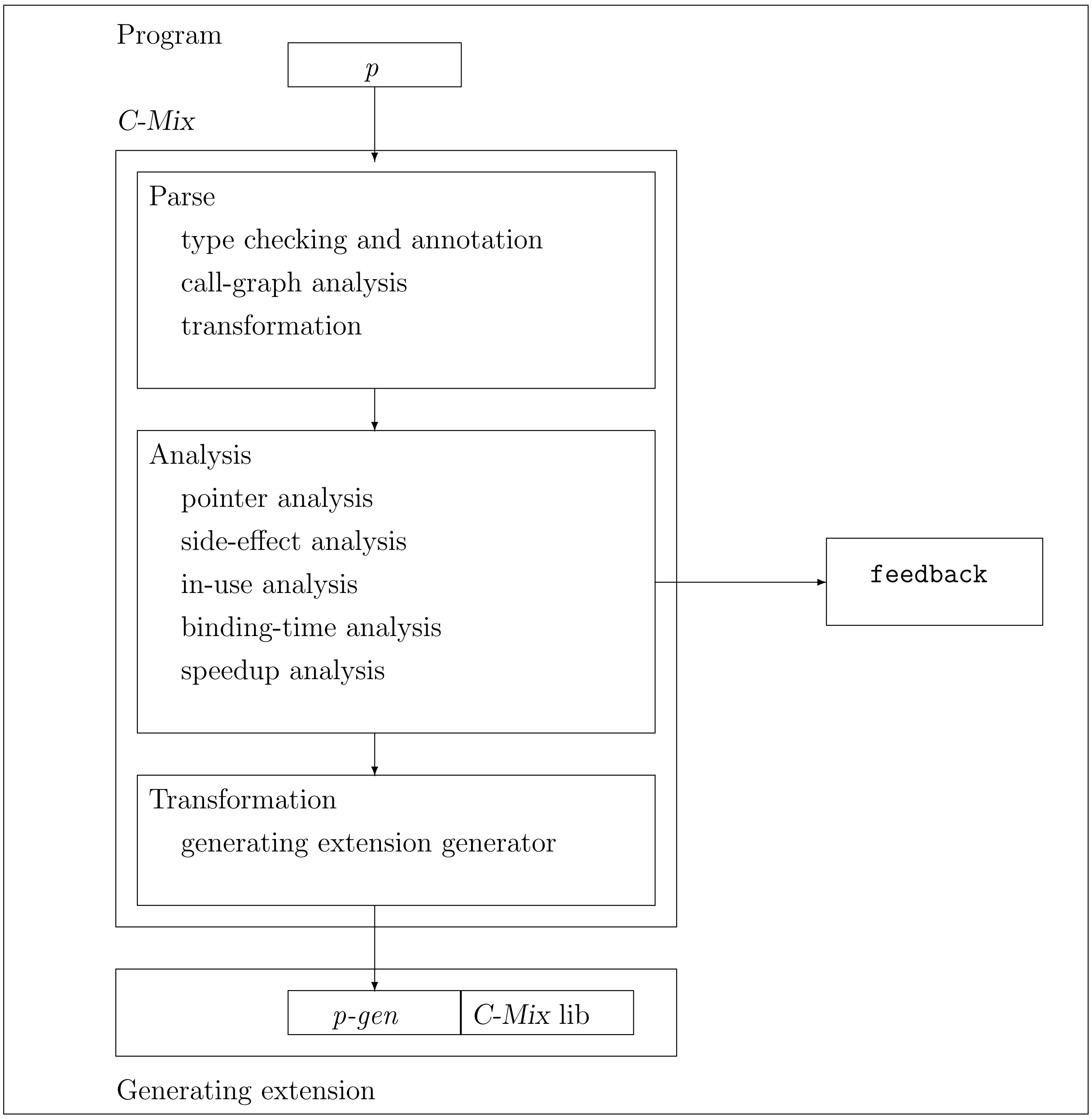

Figure 4: Overview of C-Mix

1.4.3 How this work differs from other work

The results reported in this thesis differ in several respects from other work. Three underlying design decisions have been: *efficiency*, *reality*, and *usability*. Consider each in turn.

Efficiency. The literature contains many examples on program analyses that need nights to analyze a 10 line program. In our opinion such analyses are nearly useless in practice.[10] From a practical point of view, it is unacceptable if *e.g.* a pointer analysis needs an hour to analyze a 1,000 line program. This implies that we are as much concerned about *efficiency* as *accuracy*. If extra precision is not likely to pay-off, we are inclined to choose efficiency. Furthermore, we are concerned with storage usage. Many analyses simply generate too much information to be feasible in practice.

Reality. A main purpose with this work is to demonstrate that semantically-based analysis and transformation of realistic languages is possible, and to transfer academic results to a realistic context. This implies that we are unwilling to give up analysis of, for instance, function pointers and casts, even though it would simplify matters, and allow more precise approximations in other cases. Moreover, we do not consider a small "toy-language" and claim that the results scale up. In our experience, "syntactic sugaring" is not always just a matter of syntax; it may be of importance whether complex expressions are allowed or not. One example, in partial evaluation feedback to the user is desirable. However, feedback in the form of some incomprehensible intermediate language is of little help.

Usability. Much information can be — and has been — inferred about programs. However, if the information makes little difference for the transformations carried out or the optimizations made possible, it is good for nothing. In the author's opinion it is important that program analyses and transformations are tested on realistic programs. This implies that if we see that an accurate analysis will be no better than a coarse but fast analysis, we trade efficiency for precision.

1.4.4 Why C?

It can be argued, and we will see, that C is not the most suitable language for automatic analysis and transformation. It has an open-ended semantics, supports machine-dependent features that conflicts with semantic-preserving transformation, and is to some extent unhandy.

On the other hand, C is one of the most pervasively used languages, and there are many C programs in the computer community that are likely to be around for years. Even though other languages in the long term are likely to replace C, we suspect this is going to take a while. Commercial software companies are certainly not willing to give up on C simply because the lambda calculus is easier to analyze.

Next, even though efficient compilers for C exist, many of these use "old" technology. One example, the Gnu C compiler does not perform any inter-procedural analysis. In this thesis we consider separate analysis and describe practical ways to realize truly separate inter-procedural analysis.

Finally, recently C has become a popular target language for high-level language compilers. Hence, there are good reasons for continued research on analysis and optimization of C programs.

[10]A reservation: if an analysis can infer very strong information about a program, it may be worthwhile.

1.5 Overview of thesis

The remainder of this thesis is organized as follows.

The first two chapters concern transformation of programs into their generating extension. In Chapter 3 we develop a *generating extension transformation* for the Ansi C language. We describe specialization of the various language constructs, and the supporting transformations. Moreover, we briefly describe a generating extension library implementing routines used in all extensions.

Chapter 4 develops an inter-procedural constraint-based "point-to" analysis for C. For every object of pointer type, it approximates the set of abstract locations the object may contain. The analysis is used by analyses to avoid worst-case assumptions due to assignments via pointers.

In Chapter 5 an efficient constraint-based binding-time analysis is developed. The analysis runs in almost linear time, and is fast in practice. An extension yields a polyvariant binding time classification.

Chapter 6 contains two analyses. A side-effect analysis identifies conditional and unconditional side-effects. Conditional side-effects is found via a control-dependence analysis. The in-use analysis computes an approximation of the objects used by a function. Both analyses are formulated as monotone data-flow problems.

Chapter 8 considers speedup in partial evaluation from a theoretical and analytical point of view. We prove that partial evaluation at most can give linear speedup, and develop a simple speedup analysis to estimate the prospective relative speedup.

Separate and incremental analysis is the subject of Chapter 7. All but trivial programs are separated into modules which render inter-procedural analysis hard. We develop in detail a separate constraint-based binding-time analysis. Next, we describe an incremental constraint solver that accommodates incremental binding-time analysis.

Chapter 9 provides several experimental result produces by *C-Mix*, and assesses the utility of the system.

In Chapter 10 we consider stronger transformations more broadly and in particular driving. Some initial steps towards automation of driving for C is taken.

Chapter 11 holds the conclusion and describes several areas for future work. Appendix 11.2 contains a Danish summary.

Chapter 2

The C Programming Language

The C programming language is pervasive in both computer science and commercial software engineering. The language was designed in the seventies and originates from the Unix operating system environment, but exists now on almost all platforms. It is an efficient, imperative language developed for the programmer; an underlying design decision was that if the programmer knows what he is doing, the language must not prevent him from accomplishing it. The International Standard Organization published in 1988 the ISO Standard which also was adopted by the Ansi standard; from now on Ansi C.

C supports recursive functions, multi-level pointers, multi-dimensional arrays, user-defined structures and unions, an address operator, run-time memory allocation, and function pointers. Further, macro substitution and separate compilation are integrated parts of the language. It has a small but flexible set of statements, and a rich variety of operators.

Both low and high-level applications can be programmed in C. User-defined structures enable convenient data abstraction, and by the means of bit-fields, a port of a device can be matched. Low-level manipulation of pointers and bytes allows efficient implementation of data structures, and machine-dependent features enable the development of operating systems and other architecture-close applications. This renders program analysis and optimization hard, but is a major reason for the language's success. Recently, C has become a popular target language for high-level language compilers, so efficient and automatic optimization of C continues to have importance.

We describe the abstract syntax of C and some additional static semantic rules that assures a uniform representation. Next, we specify the dynamic semantics by means of a structural operational semantics that comply to the Standard.

In high-level transformation systems, comprehensible feedback is desirable. We describe a representation of programs that accommodates feedback, and is suitable for automatic program analysis and transformation. In particular, we consider separation of struct and union definitions convenient when doing type-based analysis. Furthermore, we define and give an algorithm for computing a program's static-call graph. The static-call graph approximates the invocation of functions at run-time, and is used by context-sensitive program analyses.

2.1 Introduction

The C programming language was designed for and implemented in the UNIX operating system by Dennis Ritchie in the seventies. It is an efficient imperative language offering features such as recursive functions, multi-level pointers, run-time memory allocation, user defined types, function pointers, and a small, but flexible, set of statements. A preprocessor is responsible for macro substitution, and separate compilation is integrated.

Currently, C is one of the most perceived and used languages in software engineering, and it exists on nearly all platforms. All signs seem to indicated that C will continue to be dominant in future professional software engineering for many years. The language is ancestor to the increasingly popular object-oriented C++ language.

2.1.1 Design motivations

Trenchant reasons for the language's success is that it is terse, powerful, flexible and efficient. For example, there are no I/O-functions such as '`read`' and '`write`' built into the language. Communication with the environment is provided via a set of standard library functions. Thus, C actually consists of two parts: a *kernel language* and a *standard library*. The library defines functions for standard input and output (*e.g.* '`printf()`'); mathematical functions (*e.g.* '`pow()`'), string manipulations (*e.g.* '`strcmp()`'), and other useful operations. Externally defined functions are declared in the standard header files. In the following we use "the C language" and "the kernel language" interchangeably unless otherwise explicitly specified.

Even though the language originally was designed for operating system implementation, its convenient data abstraction mechanism enables writing of both low and high-level applications. Data abstraction is supported by user-defined types. By the means of bit-fields, members of structs can be laid out to match the addresses and bits of *e.g.* hardware ports. Several low-level operations such as bit shifts, logical 'and' and other convenient operations are part of the language. Further, casts of pointers to "void pointers" allows writing of generic functions.

The spirit of the language is concisely captured in the motivations behind the Standard, as stated in the "Rationale for the ANSI — programming language C" [Commitee 1993]:

- Trust the programmer.
- Don't prevent the programmer from doing what needs to be done.
- Keep the language small and simple.
- Provide only one way to do an operation.
- Make it fast, even if it is not guaranteed to be portable.

An example: C is not strongly-typed; integers can be converted to pointers, opening up for illegal references. Years of experience have, however, demonstrated the usefulness of this so it was not abandoned by the Standard. Instead the Standard specifies that such conversions must be made explicit. When the programmer knows what he or she is doing it shall be allowed.

The C language has some deficiencies, mostly due of the lack of strong-typing to prevent type errors at run time [Pohl and Edelson 1988].

2.1.2 Conforming and strictly-conforming programs

A substantial milestone in the language's history was 1988 where the International Standard Organization ISO published the ISO C Standard document [ISO 1990,Schildt 1993]. The Standard was modeled after the reference manual published by "*the C programming language*" [Kernighan and Ritchie 1988], and tried to preserve the spirit of the language. The Standard was also adopted by the American standard ANSI.

A program can comply the Standard in two ways: it can be *conforming* or *strictly conforming*. A strictly conforming program shall only use features described in the Standard, and may not produce any result that depends on undefined, unspecified or implementation-defined behavior. A conforming program may rely on implementation-defined features made available by a conforming implementation.

Example 2.1 Suppose that '`struct S`' is a structure, and consider the code below.

```
extern void *malloc(size_t);
struct S *p = (struct S *)malloc(sizeof(struct S));
```

This code is conforming, but not strictly conforming: a "struct S" pointer has stricter alignment requirement than a "void" pointer, and the `sizeof` operator is implementation-defined. **End of Example**

The intension is that strictly conforming programs are independent from any architectural details, and henceforth highly portable. On the other hand, implementation of for instance operating systems and compilers do require implementation-dependent operations, and this was therefore made part of the Standard.

Example 2.2 The result of a cast of a pointer to an integral value or vice versa is implementation defined. A cast of a pointer to a pointer with less alignment requirement, *e.g.* a "void" pointer, is strictly conforming. The evaluation order of function arguments is unspecified. Hence, neither a strictly nor a conforming program may rely on a particular evaluation order. **End of Example**

In this thesis we are only concerned with implementation- and architecture transparent analysis and transformation. Thus, program analyses shall not take peculiarities of an implementation into account, and transformations shall not perform optimization of non-strictly conforming constructs, *e.g.* replace '`sizeof(int)`' by 4. Restriction to strictly conforming programs is too limited for realistic purposes. Since all but trivial programs are separated into modules, we shall furthermore not restrict our attention to monolithic programs.

In general, no non-strictly conforming parts of a program shall be optimized, but will be suspended to run-time. Moreover, analyses shall not be instrumented with respect to a particular implementation.

2.1.3 User feedback

Program specialization by partial evaluation is a fully-automated high-level program optimization technique, that in principle requires no user guidance nor inspections. However, program specialization is also an ambitious software engineering process with goals that go far beyond traditional program optimization. Sometimes these goals are not met, and the user (may) want to know *what* went wrong.

The situation is similar to type checking. When a program type checks, a compiler needs not output any information. However, in the case of type errors, some informative error messages which ideally indicate the reason toe the error are desirable. Possible feedback includes:

- Output that clearly shows where it is feasible to apply specialization, *e.g.* binding-time annotations and profile informations.
- Information that describes likely gains; *i.e.* prospective speedups and estimates of residual program size.
- General feedback about the program's dynamic properties, *e.g.* side-effects, call information and the like.
- The source of dynamic (suspended) computations.

Naturally, the information must be connected to the subject program. Feedback in the form of intermediate, machine constructed representations is less useful. This renders probably "three address code" [Aho *et al.* 1986] useless for user feedback.

2.1.4 Program representation

Most previous work has used intermediate program representations where complex constructs are transformed into simpler ones. For example, an assignment '`s->x = 2`' is simplified to the statements '`t = *s; t.x = 2`'. This is not an option when program related feedback must be possible. Furthermore, unrestrained transformation may be semantically wrong. For example, "simplification" of '`a && b`' into '`(a != 0) * (b != 0)`' eliminates a sequence point and demands evaluation of '`b`'.

In this thesis we use a program representation closely resembling the structure of the subject program. The advantages include:

- Transformation into a simpler intermediate form, *e.g.* static single assignment form, tends to increase the size of the subject program. We avoid this.
- User feedback can be given concisely; essentially, the user sees nothing but the subject program.
- Simplification of constructs may throw away information. For example, it may be useful information that '`a`' in '`a[2]`' points to an array, which is not apparent from the equivalent '`*(a + 2)`'.

Some disadvantages are less control-flow information, and more cases to consider.

In the *C-Mix* system, the user receives graphical feedback in the form of annotated programs. For instance, the binding time separation of a program is illustrated by dynamic constructs in bold face.

2.1.5 Notation and terminology

In this thesis we use the notation and terminology as defined by the Standard, and, secondly, used in the C community.

A variable declaration that actually allocates some storage is called a *definition*. A declaration that simply brings a name into scope is called a *declaration*. For example, '`int x`' is a definition, and '`extern int x`' is a declaration. Since we are going to use the term "dynamic" for "possibly unknown", we use the term "run-time memory allocation" as opposed to "dynamic memory allocation".

A program (usually) consists of several translation units.[1] A program is one or more translation units where one of them contains a '`main()`' function. Identifiers defined in modules other than the one being considered are *external*. Whether a function or variable is external depends solely on the existence of a definition in the file. If a definition exists, the function or variable is non-external.

2.1.6 Overview of the chapter

The chapter is structured as follows. In Section 2.2 we describe an abstract syntax for C. Section 2.3 is concerned with representation of programs and defines static-call graphs. Section 2.4 provides an operational semantics for abstract C. Section 2.5 mentions related work, and Section 2.6 concludes.

2.2 The syntax and static semantics of C

This section presents an abstract syntax for C. The abstract syntax is deliberately formed to resemble concrete C, but does contain some simplifications. For example, all type definitions must appear at the top of a translation unit; variable and function definitions cannot be mixed; and initializers cannot be provided as a part of a definition.

2.2.1 Syntax

The abstract syntax of C is given in the Figures 5 and 6. The former defines the syntax of type definitions, global variable and function definitions, and the latter defines statements and expressions.

A *translation unit* consists of three parts. First come type definitions, followed by declaration and definition of global variables and external functions, and lastly come function definitions.

[1]A translation unit roughly corresponds to a file.

const	$\in \mathcal{C}onstants$	Constants
id	$\in \mathcal{I}d$	Identifiers
label	$\in \mathcal{L}abel$	Labels
op	$\in \mathcal{O}p$	Operators
translation-unit	::= type-def* decl* fun-def*	Module
type-def	::= `struct` *id* { strct-decl$^+$ }	Type definition
	$\mid$ `union` *id* { strct-decl$^+$ }	
	$\mid$ `enum` *id* { enum* }	
strct-decl	::= decl $\mid$ decl : exp	Struct declarator
enum	::= *id* $\mid$ *id* = exp	Enumerator declarator
decl	::= storage-spec* *id* type-spec	Declaration
storage-spec	::= `extern` $\mid$ `register`	Storage specifier
type-spec	::= `void` $\mid$ `char` $\mid$ `int` $\mid$ `double` $\mid$ . . .	base type
	$\mid$ `struct` *id* $\mid$ `union` *id*	Struct type
	$\mid$ `enum` *id*	Enum type
	$\mid$ * type-spec	Pointer type
	$\mid$ `[`*const*`]` type-spec	Array type
	$\mid$ (declaration*) type-spec	Function type
fun-def	::= type-spec *id*`(` declaration*`)` body	Function definition
body	::= { decl* stmt* }	Function body

Figure 5: Abstract syntax of C (part 1)

A *type definition* can introduce a struct, a union or an enumeration type. Type names introduced by the means of '`typedef`' are syntax and henceforth not present in the abstract syntax. Similar remarks apply to forward declarations. The abstract syntax supports bit-fields, and enumeration constants can be given specific values. Since enumeration constants to a large extent act as constants we often regard these as "named constants".[2] Struct and union definitions are not nested.

A *declaration* has three parts: a list of *storage specifiers*, an *identifier*, and a list of *type specifiers*. For our purposes the storage specifier '`extern`' suffices. Other specifiers such as '`auto`', '`static`', or '`register`' are not used in this thesis, but can be added if convenient.[3]

A *type specifier* is a list of specifiers ending with a base, struct[4] or enumeration specifier. A type specifier shall be read left to right. To readability, we surround type specifiers by brackets, *e.g.* we write the concrete type '`int *`' as $\langle * \rangle \, \langle \texttt{int} \rangle$.

Example 2.3 The following examples show the connection between some concrete C declarations and the corresponding abstract syntax notations.

[2]The revision of the Standard is likely to eliminate the implicit type coercion between enumerators and integers, thus requiring strong typing of enumeration types.

[3]Of course, the implementation correctly represents all storage specifiers but they are ignored except by the unparser.

[4]In the rest of this thesis we use 'struct' to mean "struct or union".

`int x`	$\mathtt{x} : \langle\mathtt{int}\rangle$
`int *x`	$\mathtt{x} : \langle*\rangle\,\langle\mathtt{int}\rangle$
`double a[10]`	$\mathtt{a} : \langle[10]\rangle\,\langle\mathtt{double}\rangle$
`char *(*fp)(int x)`	$\mathtt{fp} : \langle*\rangle\,\langle(\mathtt{x} : \langle\mathtt{int}\rangle)\rangle\,\langle*\rangle\,\langle\mathtt{char}\rangle$

A type specifier cannot be empty. (The last definition specifies a pointer to a function taking an integer argument and returning a pointer to a char.) **End of Example**

Given a declaration '`extern` $\mathtt{x} : \langle*\rangle\,\langle\mathtt{int}\rangle$', '`x`' is called the *declarator*, '`extern`' the *storage specifier*, and '$\langle*\rangle\,\langle\mathtt{int}\rangle$' the *type*.

The type qualifiers '`const`' and '`volatile`' are not used in this thesis, and are therefore left out of the abstract syntax. The qualifiers can be introduced as "tags" on the base and pointer type specifiers, if so desired.

A *function definition* consists of a type, an identifier, a parameter list, and a function body. A function body is a list of local (automatic) variable definitions[5] and a sequence of statements.

A *statement* can be the empty statement, an expression statement, a conditional statement, a loop statements, a labeled statement, a jump, a function return, or a block. Notice that blocks do not support local definitions. For simplicity, all automatic variables are assumed to have function scope.[6]

In the following we will often assume that '`break`' and '`continue`' are replaced by '`goto`' to make the control flow explicit. Notice that explicit control flow easily can be added to '`break`' and '`continue`' statements, so this is solely for notational purposes.

An *expression* can be a constant, a variable reference, a struct indexing, a pointer dereference, an application of a unary or binary operator, function call, assignment, or pre- and post increment operators. We often use increment as example operator. The special call '`alloc()`' denotes run-time memory allocation. Given a type name T, '`alloc(`T`)`' returns a pointer to an object suitable for representation of T objects. The introduction of this special form is motivated in Chapter 3. For now we will assume that '`alloc()`' is used/inserted by the user. For ease of presentation, the unary operators '`*`' (pointer dereference) and '`&`' (address operator) are treated as special forms.

We differentiate between the following types of call expressions: calls to '`extern`' functions (*e.g.* '`pow(2,3)`'); direct calls to (module-)defined functions (*e.g.* '`foo(2)`'); and calls via function pointers (*e.g.* '`(*fp)(13)`'). In the following we indicate calls of the former category by '*ef*`()`'; direct calls to user defined functions by '*f*`()`'; and indirect calls by 'e_0`()`'. The three kinds can be differentiated purely syntactically; notice that function pointers may point to both defined and external functions.

2.2.2 Static semantics

The static semantics of C is defined by the Standard [ISO 1990]. We impose some additional requirements for the sake of uniformity.

[5]Notice, *definitions*, not *declarations*!

[6]The main reason for this restriction is given in Chapter 3.

stmt	::=	;	Empty statement
	|	exp	Expression
	|	`if` `(` exp `)` stmt `else` stmt	If-else
	|	`switch` `(` exp `)` stmt	Multi-if
	|	`case` exp : stmt	Case entry
	|	`default` : stmt	Default entry
	|	`while` `(` exp `)` stmt	While loop
	|	`do` stmt `while` `(` exp `)`	Do loop
	|	`for` `(` exp ; exp ; exp `)` stmt	For loop
	|	*label* : stmt	Label
	|	`goto` *label*	Jump
	|	`break` | `continue`	Loop break/continue
	|	`return` | `return` exp	Function return
	|	{ stmt* }	Block
exp	::=	*const*	Constant
	|	*id*	Variable
	|	exp `.` *id*	Struct index
	|	`*`exp	Pointer dereference
	|	exp`[`exp`]`	Array index
	|	`&`exp	Address operator
	|	*op* exp	Unary operator
	|	exp *op* exp	Binary operator
	|	`alloc` (type)	Runtime allocation
	|	*id* (exp*)	Extern function call
	|	*id* (exp*)	User function call
	|	exp (exp*)	Indirect call
	|	`++`exp | `--`exp	Pre-increment
	|	exp`++` | exp`--`	Post-increment
	|	exp *aop* exp	Assignment
	|	exp , exp	Comma expressions
	|	`sizeof` (type)	Sizeof operator
	|	(type-spec) exp	Cast

Figure 6: Abstract syntax of C (part 2)

- External functions shall be declared explicitly.[7]
- Array specifiers in parameter definitions shall be changed to pointer specifiers.
- Functions shall explicitly `return` (a value, in the case of non-void functions).
- Switches shall have a `default` case.
- Optional expressions are not allowed (can be replaced by a constant, say).
- Array index expressions are arranged such that e_1 in e_1`[`e_2`]` is of pointer type.
- The type specifier '`short int`' shall be specified '`short`' and similar for '`long`'. This implies that base types can be represented by one type specifier only.

[7]Defined functions should not be declared, of course.

- The storage specifier '`extern`' shall only be applied to global identifiers.
- Conversions of function designators must be explicit, *i.e.* the address of a function is taken by '`fp = &f`', and an indirect call is written '`(*fp)()`'.
- Overloading of operators must be resolved, *e.g.* '`-`' applied to pointers shall be syntactically different from '`-`' applied to integers.

Notice that all the conditions are purely syntactical, and can be fulfilled automatically (*e.g.* during parsing). For example, `return` statements can be inserted in '`void`' functions.

2.2.3 From C to abstract C

The structure of abstract syntax closely resembles concrete C, allowing informative feedback. The syntactic omissions include the '`static`' specifier, the possibility of definitions inside statement blocks and conditional expressions.[8]

The storage specifier '`static`' has two interpretations. When applied to a global definition, the definition gets file scope. When applied to a local definition, the variable is statically allocated. By renaming of static identifiers and by lifting local (static) definitions to the global level, file scope and static allocation, respectively, can be assured.[9]

Local definitions can be lifted to the function level by renaming of identifiers. This transformation may introduce superfluous local variables, but these can be removed by simple means.

Finally, a conditional expression 'e_1`?` e_2`:`e_3' in an evaluation context context $E[\,]$ can be replaced by introduction of a new local variable '`x`', and the transformation

```
if (e_1) x = e_2 else x = e_3; E[x]
```

where '`x`' has same type as e_1. This transformation is non-trivial; if the evaluation context contains sequence points, *e.g.* a comma expression, the context must be broken down to preserve evaluation order.

2.3 A representation of C programs

In this section we outline both an abstract and a concrete representation of (abstract) C. Unlike other approaches we do not need a complicated representation, *e.g.* static single-assignment representation or the like. Essentially, an *abstract syntax tree* and a *static call graph* suffice.

In the last part of the section we consider some deficiencies regarding "sharing" of user defined types in connection with type-based analysis, and describe methods for alleviating these.

[8]The remaining constructs such as the 'register' specifier and 'const' qualifier can be added to the abstract syntax as outlined in the previous section.

[9]For other reasons it may be desirable to include the 'static' specifier in the abstract syntax. For example, an analysis might exploit the knowledge that a function is local.

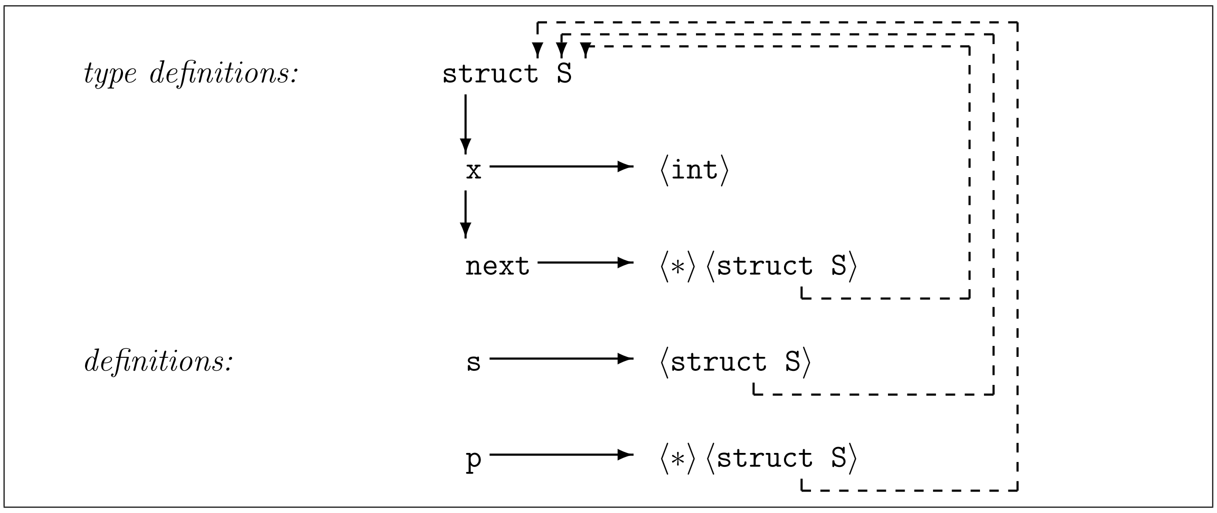

Figure 7: Representation of some definitions

2.3.1 Program representation

A program p is denoted by a triple $\langle \mathcal{T}, \mathcal{D}, \mathcal{F} \rangle$ of a set $\mathcal{T}$ of type definitions, a set $\mathcal{D}$ of global variable definitions and declarations, and a set $\mathcal{F}$ of function definitions. No function definitions shall be contained in $\mathcal{D}$.

A function definition consists of a quadruple $\langle T, D_p, D_l, B \rangle$ of a return type T, sets D_p and D_l of parameter and local variable definitions, and the representation B of the statement sequence making up the function body. For example, the function

```
int id(int x)
{
   return x;
}
```

is abstractly described by $\langle \texttt{int}, \{x : \langle \texttt{int} \rangle\}, \{\}, \{S_{return}\} \rangle$ where S_{return} is the representation of the `return` statement.

A function body is represented as a single-exit control-flow graph $\langle S, E, s, e \rangle$, where S is a set of *statement nodes*, E is a set of *control-flow edges*, and s, e are unique *start* and *exit* nodes, respectively [Aho *et al.* 1986]. The end node e has incoming edges from all `return` statements. The start node s equals the initial statement node.

A declaration (definition) is represented by the declarator node and a list of types. We denote a type using a "bracket" notation, *e.g.* '`int x`' is written `x` : ⟨`int`⟩.

A type definition is a list of member declarations. Figure 7 gives illustrative examples.

Example 2.4 Figure 7 depicts the representation of the definitions below.

```
struct S { int x; struct S *next; } s, *p;
```

Section 2.3.3 below elaborates on the representation of structure types. **End of Example**

Expressions are represented by their abstract syntax tree.

2.3.2 Static-call graph

To conduct inter-procedural — or context-sensitive — program analysis, *static-call graphs* are useful. A static-call graph abstracts the invocations of functions, and is mainly employed to differentiate invocations of functions from different contexts.

A function call is static or dynamic depending on the binding time of the function designator. A call '$f(\ldots)$' is classified as *static*, since the name of the called function is syntactically (statically) given. An indirect call '$e(\ldots)$' is classified as *dynamic*, since the function invoked at run-time cannot (in general) be determined statically.[10] A static-call graph represents the invocations of functions due to static calls. Indirect calls are not represented. Assume that all static calls are labeled uniquely.

A function may be called from different call-sites, *e.g.* if there are two calls '$\texttt{foo}^1\texttt{()}$' and '$\texttt{foo}^2\texttt{()}$', function '`foo()`' is invoked in the *contexts* of calls 1 and 2. Each call gives rise to a *variant* of the (called) function, corresponding to the context of the call.

Definition 2.1 *A* static-call graph $\mathcal{SCG}$: *CallLabel* $\times$ *Variant* $\to$ *Id* $\times$ *Variant maps a call label and a variant number into a function name and a variant number, such that* $\mathcal{SCG}(l,m) = \langle f, n\rangle$ *if call-site* l *in the* m*'th variant of the function containing call-site* l *invokes variant* n *of* f. □

Suppose that call-site 1 appears in '`main()`', and function '`foo()`' is called. Then $\mathcal{SCG}(1,1) = \langle \texttt{foo}, 2\rangle$ represents that variant 1 of '`main()`' calls variant 2 of '`foo()`'.

Example 2.5 Consider the following program.

```
int main(void)                   int foo(void)        int bar(void)
{                                {                    {
   int (*fp)(void) = &bar();        return bar³();       return 1;
   foo¹();                       }                    }
   foo²();
   (*fp)();
   return 0;
}
```

The static-call graph is shown in Figure 8. Notice that the indirect call is not represented.

The tabulation of $\mathcal{SCG}$ is show below,

Call\Variant	1	2
1	$\langle$foo, 1$\rangle$	
2	$\langle$foo, 2$\rangle$	
3	$\langle$bar, 1$\rangle$	$\langle$bar, 2$\rangle$

where the call-sites refer to the program above. **End of Example**

Suppose a program contains recursive or mutually recursive functions. This implies that an infinite number of function invocation contexts exist (at run-time). To represent this, a *1-limit* approximation is adopted.

[10]Naturally, an analysis can compute a set of functions that possibly can be invoked, see Chapter 4.

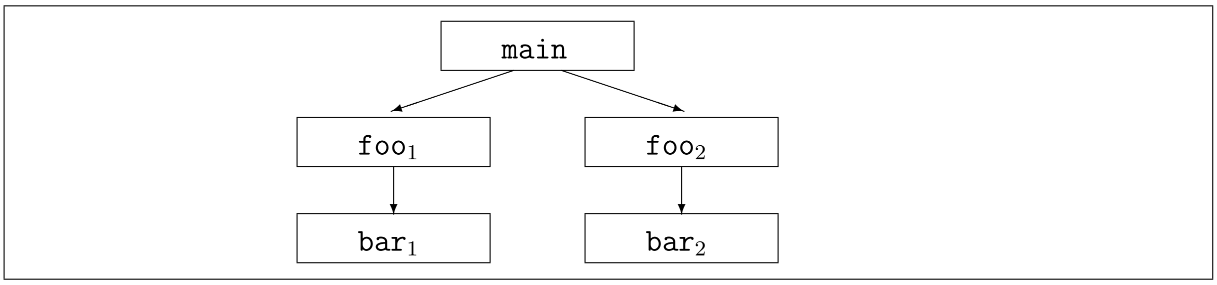

Figure 8: Static-call graph for Example 2.5

Definition 2.2 *The number of times a variant of a recursive function is created due to the (direct or indirect) recursive call is called the k-limit.* □

The use of 1-limit means that a recursive call to a function is approximated by the same context as the first invocation of the function.

Example 2.6 The static-call graph for the (useless) function

```
int rec(int x) { return rec$^1$(x) }
```

is given by $\mathcal{SCG}(1,1) = \langle \text{rec}, 1\rangle$. **End of Example**

Better precision is obtained by larger k-limits. Even though our implementation supports arbitrary k-limits, we have not experimented with this. A static-call graph can be computed by Algorithm 2.1.

Algorithm 2.1 *Computation of static-call graph.*

```
invoke(main,1);
/* invoke: build scg for variant 'var' of function 'fun' */
void invoke(fun, var)
{
   for ( $f^l$() in StaticCalls(fun)) {
      if ( v = stack.seenB4(<$f$,var>,K_LIMIT) )
         /* approximate with variant v */
         scg.insert( <$l$,var> = <$f$,v> );
      else {
         /* create new variant and invoke it */
         v = next_variant($f$);
         stack.push( <$f$,v> );
         scg.insert( <$l$,var> = <$f$,v> );
         invoke($f$,v);
         stack.pop();
      }
}
```

The variable '`stack`' is a stack of function identifiers, variants. The method '`seenB4()`' scans the stack to see whether a function has been invoked '`K_LIMIT`' number of times before. In the affirmative case, the variant number is returned. Variant 0 is reserved for a special purpose, see Chapter 4. □

$$\begin{tikzcd}
\texttt{a:}\ \langle * \rangle \langle [10] \rangle \langle \texttt{int} \rangle & & & & \texttt{x:}\ \langle \texttt{int} \rangle \\
& & & \texttt{=:}\ \langle \texttt{int} \rangle \arrow[dl] \arrow[dr] & \\
& & \texttt{[]:}\ \langle \texttt{int} \rangle \arrow[dl] & & \texttt{x:}\ \langle \texttt{int} \rangle \arrow[uu] \\
& \texttt{*:}\ \langle [10] \rangle \langle \texttt{int} \rangle \arrow[dl] & & & \\
\texttt{a:}\ \langle * \rangle \to \langle [10] \rangle \langle \texttt{int} \rangle \arrow[uuuu] & & & &
\end{tikzcd}$$

Figure 9: Representation of declarations and an expression ‘`(*a)[2] = x`’

Notice that the number of variants of a function may grow exponentially, but in practice this seldom happens. Strategies for preventing this has been discussed in the literature [Cooper *et al.* 1993]. We will not consider this issue here.

A remark. Our definition of static-call graph differs from the usual notion of call graphs [Aho *et al.* 1986]. Given a program, a static-call graph approximates the *invocations* of functions at run-time starting from the ‘`main()`’ function. Traditional call graphs simply represent the set of functions a function may call.

Example 2.7 Consider the program in Example 2.5. A call graph (in the usual sense) shows that ‘`foo()`’ calls ‘`bar()`’. The static-call graph represents that variant 1 of ‘`foo()`’ calls variant 1 of ‘`bar()`’, and similarly for variant 2. **End of Example**

2.3.3 Separation of types

We shall assume that each expression is attributed with the type of the expression.

Example 2.8 The representation of ‘`int (*a)[10], x`’ and the assignment ‘`(*a)[2] = x`’ is shown in Figure 9. In principle, a declarator’s type specifier lists can be shared by expression nodes, such that an expression points to the relevant part of the declarator’s type specifier list. Since we are going to associate value flow information with specifiers, this would result in a less precise representation, *e.g.* information associated with different occurrences of a variable would be merged, so we refrain from this. **End of Example**

In type-based analysis it is convenient to associate value flow information with type specifiers. For example, the binding-time analysis of Chapter 5 assign a *binding time flag* to each type specifier.

In the case of non-struct type, a “new” type specifier list is assigned to each definition. For example, given the definitions ‘`int x, y`’, the specifier ⟨`int`⟩ is not shared between ‘`x`’ and ‘`y`’; a fresh instance is created for each.

Suppose that the struct definition

```
struct Node { int x; struct Node *next; }
```

and the definitions ‘`struct Node s,t`’ are given. Typically, the information associated with the types of members determines the lay-out of structs in memory, the appearance of a residual struct definition *etc.*

A consequence of this is that completely unrelated uses of a struct type may influence each other. For example, in the case of binding-time analysis, if ‘`s.x`’ is assigned a dynamic value, ‘`t.x`’ becomes dynamic as a “side-effect”, since the definition of the type is “shared”. Highly unfortunately if ‘`s`’ and ‘`t`’ are otherwise unrelated.

To avoid this, fresh *instances* of struct definitions can be created for each use. For example, an instance ‘`struct S1`’ can be created for ‘`s`’ and similarly an instance ‘`struct S2`’ can be assigned as type of ‘`t`’.

Example 2.9 In Chapter 3 we develop a program transformation that outputs a C program, and consequently also struct definitions. Suppose that the program contains an assignment ‘`t = s`’. Since C uses type equivalence by name, the types of ‘`s`’ and ‘`t`’ must be equal. This means that ‘`s`’ and ‘`t`’ must be given the *same* instance of the underlying struct type, *e.g.* ‘`struct S1`’.

A *value-flow analysis* can accomplish this by a single traversal of the program where the (struct) type instances in assignments, parameters assignments and function returns are union-ed. **End of Example**

To accommodate recursive struct definitions, a k-limit approximation is adopted. That is, k instances of a recursive struct definition are created. In our implementation, we use a 1-limit, since recursive structs are likely to be used in the “same” context. Larger k-limits give better precision.

2.4 Dynamic semantics

The dynamic semantics of abstract C is specified by the means of a big-step structural operational semantics [Plotkin 1981,Hennessy 1990,Nielson and Nielson 1992b]. The aim of this section is not to compete with the Standard [ISO 1990], but to provide a concise description that can serve as a foundation for specifying and reasoning about program analyses and transformations. We do not, for instance, specify the behaviour of standard-library functions and leave unmentioned many of the implicit conversions that take place.

The specification comes in four parts: inference systems for functions, statements, expressions and declarations. Further, we informally describe the modeling of the store. Precise description of memory management is not relevant for this exhibition, and moreover depends on an implementation. As usual, the store is assumed to be a consecutive set of locations where a separate part constitutes a program stack. Heap-allocation is performed in an isolated part of the store.

The semantics assigns meaning to programs — not translation units. Thus, in principle, no external function calls exist. For convenience, we shall regard calls to library functions as 'extern' calls, and use a pre-defined function $F : \mathrm{Id} \times \mathrm{Value}^* \times \mathrm{Store} \to \mathrm{Value} \times \mathrm{Store}$ to describe their behaviour. For example, $F(\mathtt{pow}, (2,3), \mathcal{S}) = (8, \mathcal{S})$. This does not include the `<setjmp.h>` functions, which we *do not* describe.

2.4.1 Notations and conventions

The set of identifiers is denoted by Id. We adopt the convention that v denotes a variable; f a user-defined function, and ef an externally defined (library) function. The set of declarations is denoted by Decl. We use d_p to range over parameter declarations and d_l to range over local definitions when it is convenient to differentiate. The set Expr is the set of expressions, and Stmt denotes the set of statements. The symbol S ranges over statements and Seq ranges over statement sequences. The set of function is denoted by Fun, and the set of types by Type.

The set of denotable values is called Value, and includes for example integers, locations, and objects of array and struct types. The set of locations is denoted by Loc.

An *environment* is a map $\mathcal{E} : \mathrm{Id} \to \mathrm{Loc}$ from identifiers to locations. The *store* is modeled as a map $\mathcal{S} : \mathrm{Loc} \to \mathrm{Value}$ from locations to values. Since we solely consider type annotated programs we omit projections on values. For instance, we write $\mathcal{S}(v)$ for $\mathcal{S}(v \downarrow \mathrm{Loc})$, when v is an address.

For simplicity we will not be explicit about errors. We courageously assume that external functions are "well-behaved", *e.g.* set the '`errno`' variable in the case of errors, and rely on *stuck* states. Infinite loops are modeled by infinite derivations. Further, we assume that output is "delivered" by writes to reserved memory locations (all taken care of by library functions).

2.4.2 Semantics of programs

Recall that a program $p = \langle \mathcal{T}, \mathcal{D}, \mathcal{F} \rangle$ is a triple of type definitions, global declarations and function definitions. Type definitions are merely interesting for static program elaboration and for storage allocation, so we ignore these below.

Definition 2.3 *Let $p = \langle \mathcal{T}, \mathcal{D}, \mathcal{F} \rangle$ be a program, and F a meaning function for library functions. Let $\mathcal{S}_0$ be a store. The meaning of the program applied to the values $v_1, \ldots, v_n$ is the value v, $[\![p]\!](v_1, \ldots, v_n) \Rightarrow v$, if*

- $\vdash^{decl} \langle d_i, 0, \mathcal{E}_{i-1}, \mathcal{S}_{i-1} \rangle \Rightarrow \langle \mathcal{E}_i, \mathcal{S}_i \rangle$ *for $d_i \in \mathcal{D}$, $i = 1, \ldots, m$*
- $\mathcal{E} = \mathcal{E}_n \circ [f_i \mapsto l_{f_i}]$, *$f_i \in \mathcal{F}$, $i = 1, \ldots, n$ where l_{f_i} are fresh locations,*
- $\mathcal{S} = \mathcal{S}_n$,
- $\mathcal{E} \vdash^{fun} \langle \mathcal{E}(f_{main}, (v_1, \ldots, v_n), \mathcal{S} \rangle \Rightarrow \langle v, \mathcal{S}' \rangle$ *where $f_{main} \in \mathcal{F}$ is the '`main`' function,*

where $\vdash^{decl}$ is defined in Section 2.4.6 and $\vdash^{fun}$ is defined in Section 2.4.3. □

$$[\text{fun}]\quad \frac{\begin{array}{l} f \equiv T\ f(d_1^p, \ldots, d_m^p)\{d_1^l, \ldots, d_n^l\ Seq\} \\ \vdash^{decl} \langle d_i^p, v_i, \mathcal{E}_i, \mathcal{S}_i \rangle \Rightarrow \langle \mathcal{E}_{i+1}, \mathcal{S}_{i+1} \rangle \\ \vdash^{decl} \left\langle d_i^l, 0, \mathcal{E}_{m+i}, \mathcal{S}_{m+i} \right\rangle \Rightarrow \langle \mathcal{E}_{m+i+1}, \mathcal{S}_{m+i+1} \rangle \\ \mathcal{E}_{m+n+1} \vdash^{stmt} \langle Seq, \mathcal{S}_{m+n+1} \rangle \Rightarrow \langle v, \mathcal{S}' \rangle \end{array}}{\mathcal{E}_1 \vdash^{fun} \langle f, (v_1, \ldots, v_m), \mathcal{S}_1 \rangle \Rightarrow \langle v, \mathcal{S}' \rangle}$$

Figure 10: Dynamic semantics for function invocation

The initial store is updated with respect to the global definitions. Next, the environment is updated to contain the addresses of functions. Finally, the value of the program is determined by the relation $\vdash^{fun}$, which is defined below.

2.4.3 Semantics of functions

A function takes a list of values and the store, and executes the statements in the context of an environment extended with the locations of parameters and local variables. This is specified by the means of the relation

$$\vdash^{fun}: \text{Loc} \times \text{Value}^* \times \text{Store} \rightarrow \text{Value} \times \text{Store}$$

defined in Figure 10. The result is a value and a (possibly) modified store.

Definition 2.4 *Let $f \in \mathcal{F}$ be a function, and $\mathcal{E}$, $\mathcal{S}$ an environment and a store, respectively. Let $v_1, \ldots, v_n$ be a list of values. The function application $f(v_1, \ldots, v_n)$ returns value v and store $\mathcal{S}'$ if*

$$\mathcal{E} \vdash^{fun} \langle f, (v_1, \ldots, v_n), \mathcal{S} \rangle \Rightarrow \langle v, \mathcal{S}' \rangle$$

where $\vdash^{fun}$ is defined in Figure 10. $\square$

Operationally speaking, the rule can be explained as follows. First, storage for parameters and local variables is allocated by the means of the $\vdash^{decl}$ relation. Parameters are initialized with the actual values v_i. Local variables are initialized with 0.[11] Finally, the statements are executed in the extended environment.

2.4.4 Semantics of statements

Execution of a statement possibly modifies the store, and transfers the control either implicitly to the next program point, or explicitly to a named (labelled) program point.[12] The semantics is specified by the means of the relation

$$\vdash^{stmt} \text{Stmt} \times \text{Store} \rightarrow \text{Value}^{\top} \times \text{Store}$$

defined in Figure 11.

[11]The Standard specifies that local variables may contain garbage so this choice adheres to the definition.

[12]Recall that we do not consider setjmp/longjmp.

[empty]	$\mathcal{E} \vdash^{stmt} \langle\ ;\ , \mathcal{S}\rangle \Rightarrow \langle \top, \mathcal{S}\rangle$
[expr]	$\dfrac{\mathcal{E} \vdash^{exp} \langle e, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle}{\mathcal{E} \vdash^{stmt} \langle e, \mathcal{S}\rangle \Rightarrow \langle \top, \mathcal{S}'\rangle}$
[if-true]	$\dfrac{\mathcal{E} \vdash^{exp} \langle e, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle \quad \mathcal{E} \vdash^{stmt} \langle S_1, \mathcal{S}'\rangle \Rightarrow \langle \nabla, \mathcal{S}''\rangle \quad v \neq 0}{\mathcal{E} \vdash^{stmt} \langle \texttt{if (}e\texttt{) } S_1 \texttt{ else } S_2, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}''\rangle}$
[if-false]	$\dfrac{\mathcal{E} \vdash^{exp} \langle e, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle \quad \mathcal{E} \vdash^{stmt} \langle S_2, \mathcal{S}'\rangle \Rightarrow \langle \nabla, \mathcal{S}''\rangle \quad v = 0}{\mathcal{E} \vdash^{stmt} \langle \texttt{if (}e\texttt{) } S_1 \texttt{ else } S_2, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}''\rangle}$
[switch-case]	$\dfrac{\mathcal{E} \vdash^{exp} \langle e, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle \quad \mathcal{E} \vdash^{stmt} \langle S_v, \mathcal{S}'\rangle \Rightarrow \langle \nabla, \mathcal{S}''\rangle \quad \texttt{case } v\texttt{: } S_v \texttt{ in } S_1}{\mathcal{E} \vdash^{stmt} \langle \texttt{switch (}e\texttt{) } S_1, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}''\rangle}$
[switch-default]	$\dfrac{\mathcal{E} \vdash^{exp} \langle e, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle \quad \mathcal{E} \vdash^{stmt} \langle S_0, \mathcal{S}'\rangle \Rightarrow \langle \nabla, \mathcal{S}''\rangle \quad \texttt{default: } S_0 \texttt{ in } S_1}{\mathcal{E} \vdash^{stmt} \langle \texttt{switch (}e\texttt{) } S_1, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}'\rangle}$
[while-true]	$\dfrac{\mathcal{E} \vdash^{exp} \langle e, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle \quad \mathcal{E} \vdash^{stmt} \langle S_1\texttt{;while (}e\texttt{) } S_1, \mathcal{S}'\rangle \Rightarrow \langle \nabla, \mathcal{S}''\rangle\, v \neq 0}{\mathcal{E} \vdash^{stmt} \langle \texttt{while (}e\texttt{) } S_1, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}''\rangle}$
[while-false]	$\dfrac{\mathcal{E} \vdash^{exp} \langle e, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle \quad v = 0}{\mathcal{E} \vdash^{stmt} \langle \texttt{while (}e\texttt{) } S_1, \mathcal{S}\rangle \Rightarrow \langle \top, \mathcal{S}'\rangle}$
[do-conv]	$\dfrac{\mathcal{E} \vdash^{stmt} \langle S_1\texttt{;while (}e\texttt{) } S_1\texttt{)}, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}'\rangle}{\mathcal{E} \vdash^{stmt} \langle \texttt{do } S_1 \texttt{ while (}e\texttt{)}, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}'\rangle}$
[for]	$\dfrac{\mathcal{E} \vdash^{stmt} \langle e_1\texttt{;while (}e_2\texttt{) \{ } S_1\texttt{;}e_3 \texttt{ \}}, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}'\rangle}{\mathcal{E} \vdash^{stmt} \langle \texttt{for (}e_1\texttt{;}e_2\texttt{;}e_3\texttt{) } S_1, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}'\rangle}$
[label]	$\dfrac{\mathcal{E} \vdash^{stmt} \langle S_1, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}'\rangle}{\mathcal{E} \vdash^{stmt} \langle l : S_1, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}'\rangle}$
[goto]	$\dfrac{\mathcal{E} \vdash^{stmt} \langle Seq_m, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}'\rangle}{\mathcal{E} \vdash^{stmt} \langle \texttt{goto } m, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}'\rangle}$
[return]	$\mathcal{E} \vdash^{stmt} \langle \texttt{return}, \mathcal{S}\rangle \Rightarrow \langle 0, \mathcal{S}\rangle$
[return]	$\dfrac{\mathcal{E} \vdash^{exp} \langle e, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle}{\mathcal{E} \vdash^{stmt} \langle \texttt{return } e, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle}$
[block]	$\dfrac{\mathcal{E} \vdash^{stmt} \langle Seq, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}'\rangle}{\mathcal{E} \vdash^{stmt} \langle \{\ Seq\ \}, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}'\rangle}$
[seq1]	$\dfrac{\mathcal{E} \vdash^{stmt} \langle S, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle}{\mathcal{E} \vdash^{stmt} \langle S; Seq, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle}$
[seq2]	$\dfrac{\mathcal{E} \vdash^{stmt} \langle S, \mathcal{S}\rangle \Rightarrow \langle \top, \mathcal{S}'\rangle \quad \mathcal{E} \vdash^{stmt} \langle Seq, \mathcal{S}'\rangle \Rightarrow \langle \nabla, \mathcal{S}''\rangle}{\mathcal{E} \vdash^{stmt} \langle S; Seq, \mathcal{S}\rangle \Rightarrow \langle \nabla, \mathcal{S}''\rangle}$

Figure 11: Dynamic semantics for statements

The set $\text{Value}^{\top} = \text{Value} \cup \{\top\}$ is employed to model the absence of a value. The element $\top$ denotes a "void" value. We use v to range over Value and ∇ to range over $\text{Value}^{\top}$.

Definition 2.5 *Let Seq be a statement sequence in a function f, and $\mathcal{E}$, $\mathcal{S}$ an environment and a store in agreement with f. The meaning of Seq is the value v and the store $\mathcal{S}'$ if*

$$\mathcal{E} \vdash^{stmt} \langle Seq, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle$$

where $\vdash^{stmt}$ is defined in Figure 11. □

[const] $\mathcal{E} \vdash^{exp} \langle c, \mathcal{S}\rangle \Rightarrow \langle \text{ValOf}(c), \mathcal{S}\rangle$

[var] $\mathcal{E} \vdash^{exp} \langle v, \mathcal{S}\rangle \Rightarrow \langle \mathcal{S}(\mathcal{E}(v)), \mathcal{S}\rangle$

[struct] $$\frac{\mathcal{E} \vdash^{lexp} \langle e_1, \mathcal{S}\rangle \Rightarrow \langle l_1, \mathcal{S}_1\rangle}{\mathcal{E} \vdash^{exp} \langle e_1.i, \mathcal{S}\rangle \Rightarrow \langle \mathcal{S}_1(l_1 + \text{Offset}(i), \mathcal{S}_1\rangle}$$

[indr] $$\frac{\mathcal{E} \vdash^{exp} \langle e_1, \mathcal{S}\rangle \Rightarrow \langle v_1, \mathcal{S}_1\rangle}{\mathcal{E} \vdash^{exp} \langle *e_1, \mathcal{S}\rangle \Rightarrow \langle \mathcal{S}(v_1), \mathcal{S}_1\rangle}$$

[array] $$\frac{\mathcal{E} \vdash^{exp} \langle e_1, \mathcal{S}\rangle \Rightarrow \langle v_1, \mathcal{S}_1\rangle \quad \mathcal{E} \vdash^{exp} \langle e_2, \mathcal{S}\rangle \Rightarrow \langle v_2, \mathcal{S}_2\rangle}{\mathcal{E} \vdash^{exp} \langle e_1\texttt{[}e_2\texttt{]}, \mathcal{S}\rangle \Rightarrow \langle \mathcal{S}_2(v_1 + v_2), \mathcal{S}_2\rangle}$$

[address] $$\frac{\mathcal{E} \vdash^{lexp} \langle e, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}_1\rangle}{\mathcal{E} \vdash^{exp} \langle \& e, \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}_1\rangle}$$

[unary] $$\frac{\mathcal{E} \vdash^{exp} \langle e_1, \mathcal{S}\rangle \Rightarrow \langle v_1, \mathcal{S}_1\rangle}{\mathcal{E} \vdash^{exp} \langle o\ e_1, \mathcal{S}\rangle \Rightarrow \langle \mathcal{O}(o)(v_1), \mathcal{S}_1\rangle}$$

[binary] $$\frac{\mathcal{E} \vdash^{exp} \langle e_1, \mathcal{S}\rangle \Rightarrow \langle v_1, \mathcal{S}_1\rangle \quad \mathcal{E} \vdash^{exp} \langle e_2, \mathcal{S}_1\rangle \Rightarrow \langle v_2, \mathcal{S}_2\rangle}{\mathcal{E} \vdash^{exp} \langle e_1\ o\ e_2, \mathcal{S}\rangle \Rightarrow \langle \mathcal{O}(o)(v_1, v_2), \mathcal{S}_2\rangle}$$

[alloc] $\mathcal{E} \vdash^{exp} \texttt{alloc(}T\texttt{)} \Rightarrow \langle l, \mathcal{S}_1\rangle \quad l$ fresh location

[extern] $$\frac{\begin{array}{l}\mathcal{E} \vdash^{exp} \langle e_1, \mathcal{S}\rangle \Rightarrow \langle v_1, \mathcal{S}_1\rangle \ldots \mathcal{E} \vdash^{exp} \langle e_n, \mathcal{S}_{n-1}\rangle \Rightarrow \langle v_n, \mathcal{S}_n\rangle \\ F(f, (v_1, \ldots, v_n), \mathcal{S}_n) = \langle v, \mathcal{S}'\rangle\end{array}}{\mathcal{E} \vdash^{exp} \langle f(e_1, \ldots, e_n), \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle}$$

[user] $$\frac{\begin{array}{l}\mathcal{E} \vdash^{exp} \langle e_1, \mathcal{S}\rangle \Rightarrow \langle v_1, \mathcal{S}_1\rangle \ldots \mathcal{E} \vdash^{exp} \langle e_n, \mathcal{S}\rangle \Rightarrow \langle v_n, \mathcal{S}_n\rangle \\ \mathcal{E} \vdash^{fun} \langle \mathcal{E}(f), (v_1, \ldots, v_n), \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle\end{array}}{\mathcal{E} \vdash^{exp} \langle f(e_1, \ldots, e_n), \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle}$$

[call] $$\frac{\begin{array}{l}\mathcal{E} \vdash^{exp} \langle e_0, \mathcal{S}\rangle \Rightarrow \langle v_0, \mathcal{S}_0\rangle \\ \mathcal{E} \vdash^{exp} \langle e_1, \mathcal{S}_0\rangle \Rightarrow \langle v_1, \mathcal{S}_1\rangle \ldots \mathcal{E} \vdash^{exp} \langle e_n, \mathcal{S}_{n-1}\rangle \Rightarrow \langle v_n, \mathcal{S}_n\rangle \\ \mathcal{E} \vdash^{fun} \langle v_0, (v_1, \ldots, v_n), \mathcal{S}_n\rangle \Rightarrow \langle v, \mathcal{S}'\rangle\end{array}}{\mathcal{E} \vdash^{exp} \langle e_0(e_1, \ldots, e_n), \mathcal{S}\rangle \Rightarrow \langle v, \mathcal{S}'\rangle}$$

[pre] $$\frac{\mathcal{E} \vdash^{lexp} \langle e_1, \mathcal{S}\rangle \Rightarrow \langle l_1, \mathcal{S}_1\rangle \quad \mathcal{S}_1 = \mathcal{S}[(\mathcal{S}(l) + 1)/l]}{\mathcal{E} \vdash^{exp} \langle \texttt{++}e_1, \mathcal{S}\rangle \Rightarrow \langle \mathcal{S}_1(l), \mathcal{S}_1\rangle}$$

[post] $$\frac{\mathcal{E} \vdash^{lexp} \langle e_1, \mathcal{S}\rangle \Rightarrow \langle l_1, \mathcal{S}_1\rangle \quad \mathcal{S}_1 = \mathcal{S}[(\mathcal{S}(l) + 1)/l]}{\mathcal{E} \vdash^{exp} \langle e_1\texttt{++}, \mathcal{S}\rangle \Rightarrow \langle \mathcal{S}(l), \mathcal{S}_1\rangle}$$

[assign] $$\frac{\mathcal{E} \vdash^{lexp} \langle e_1, \mathcal{S}\rangle \Rightarrow \langle l_1, \mathcal{S}_1\rangle \quad \mathcal{E} \vdash^{exp} \langle e_2, \mathcal{S}_1\rangle \Rightarrow \langle v_2, \mathcal{S}_2\rangle}{\mathcal{E} \vdash^{exp} \langle e_1 = e_2, \mathcal{S}\rangle \Rightarrow \langle v_2, \mathcal{S}_2[v_2/l_1]\rangle}$$

[comma] $$\frac{\mathcal{E} \vdash^{exp} \langle e_1, \mathcal{S}\rangle \Rightarrow \langle v_1, \mathcal{S}_1\rangle \quad \mathcal{E} \vdash^{exp} \langle e_2, \mathcal{S}_1\rangle \Rightarrow \langle v_2, \mathcal{S}_2\rangle}{\mathcal{E} \vdash^{exp} \langle e_1, e_2, \mathcal{S}\rangle \Rightarrow \langle v_2, \mathcal{S}_2\rangle}$$

[sizeof] $\mathcal{E} \vdash^{exp} \langle \texttt{sizeof(}T\texttt{)}, \mathcal{S}\rangle \Rightarrow \langle \text{Sizeof}(T), \mathcal{S}\rangle$

[cast] $$\frac{\mathcal{E} \vdash^{exp} \langle e_1, \mathcal{S}\rangle \Rightarrow \langle v_1, \mathcal{S}_1\rangle}{\mathcal{E} \vdash^{exp} \langle \texttt{(}T\texttt{)}e_1, \mathcal{S}\rangle \Rightarrow \langle v_1, \mathcal{S}_1\rangle}$$

Figure 12: Dynamic semantics for expressions

The empty statement has no effect, and the value of an expression statement is discarded. The store may, however, be modified.

An `if` statement executes the then-branch if the expression evaluates to a value different from 0. Otherwise the else-branch is executed. In the case of a `switch`, the corresponding `case` entry is selected if any match, otherwise the `default` rules is chosen.

The rule for `while` executes the body while the test is non-zero. The `do` and `for` statements are assigned meaning by transformation into semantically equivalent `while` loops [Kernighan and Ritchie 1988, Appendix A].

A label is ignored, and in the case of a `goto`, the control is transferred to the corresponding statement sequence. A `return` statement terminates the execution with the value of the expression (and zero otherwise). The statements in blocks are executed in sequence until a `return` statement is met.

2.4.5 Semantics of expressions

Evaluation of an expression may modify the store. Actually, in C, expressions are the sole construct that can affect the store. The evaluation is described by the means of the relations

$$\vdash^{exp}, \vdash^{lexpr} \text{Expr} \times \text{Store} \rightarrow \text{Value} \times \text{Store}$$

defined in Figure 12 and 13. The former "computes" values, the latter lvalues. For simplicity we ignore that "void" functions return no value.

Definition 2.6 *Let $e \in Expr$ be an expression in a function f, executed in the environment $\mathcal{E}$ and store $\mathcal{S}$. The meaning of e is the value v and store $\mathcal{S}'$ if*

$$\mathcal{E} \vdash^{exp} \langle e, \mathcal{S} \rangle \Rightarrow \langle v, \mathcal{S}' \rangle$$

where $\vdash^{exp}$ is defined in Figure 12 and 13. □

The semantic value of a syntactic constant is given by the meaning function ValOf. The value of a variable is looked up in the store through the environment. Notice that in the case of an array variable reference, an implicit conversion from "array of T" to "pointer to T" takes place. This is not shown.

To access the value of a field, the struct object's address is calculated and the member's offset added. In the case of pointer dereference or an array indexing, the store is accessed at the location the subexpression evaluates to. Recall that array index expressions are assumed arranged such that the subexpression e_1 is of pointer type. The address operator is assigned meaning via the lvalue relation.

The rules for unary and binary operator applications use the (unspecified) semantic function O. Recall that overloading is resolved during parsing. The rule of '`alloc()`' returns the address of a free, consecutive block of memory of suitable size. External function calls, *i.e.* calls to library functions, use the F meaning function.

A function call is evaluated as follows. First, the arguments (and possibly the function expression) are evaluated. Next the return value is determined by the means of the $\vdash^{fun}$ relation.

$$
\begin{array}{ll}
\text{[var]} & \mathcal{E} \vdash^{lexp} \langle v, \mathcal{S} \rangle \Rightarrow \langle \mathcal{E}(v), \mathcal{S} \rangle \\
\text{[struct]} & \dfrac{\mathcal{E} \vdash^{exp} \langle l_1, \mathcal{S} \rangle \Rightarrow \langle l_1, \mathcal{S}_1 \rangle}{\mathcal{E} \vdash^{lexp} \langle e_1.i, \mathcal{S} \rangle \Rightarrow \langle l_1 + \text{Offset}(i), \mathcal{S}_1 \rangle} \\
\text{[indr]} & \dfrac{\mathcal{E} \vdash^{exp} \langle e_1, \mathcal{S} \rangle \Rightarrow \langle v_1, \mathcal{S}_1 \rangle}{\mathcal{E} \vdash^{exp} \langle *e_1, \mathcal{S} \rangle \Rightarrow \langle v_1, \mathcal{S}_1 \rangle} \\
\text{[array]} & \dfrac{\mathcal{E} \vdash^{exp} \langle e_1, \mathcal{S} \rangle \Rightarrow \langle v_1, \mathcal{S}_1 \rangle \quad \mathcal{E} \vdash^{exp} \langle e_2, \mathcal{S}_1 \rangle \Rightarrow \langle v_2, \mathcal{S}_2 \rangle}{\mathcal{E} \vdash^{lexp} \langle e_1 \texttt{[} e_2 \texttt{]}, \mathcal{S} \rangle \Rightarrow \langle e_1 + e_2, \mathcal{S}_2 \rangle}
\end{array}
$$

Figure 13: Dynamic semantics for left expressions

Pre and post increment expressions update the location of the subexpression, and an assignment updates the lvalue of the left hand side expression. Comma expressions are evaluated in sequence, and the rule for `sizeof` uses the semantic function SizeOf. The rules for cast is trivial (since conversions are left implicit).

The rules for computation of an expression's lvalue are straightforward. Notice that since function identifiers are bound in the environment to locations, the lvalue of an expression '`&f`' correctly evaluates to the location of '`f`'.

2.4.6 Semantics of declarations

The dynamic semantics for declarations amount to storage allocation. External declarations simply update the environment with the address of the function, which we assume available. This is necessary since the address of library functions may be taken. Definitions of global variables allocate some storage and update the environment to reflect this. This is expressed by the relation

$$\vdash^{decl}: \text{Decl} \times \text{Value} \times \text{Env} \times \text{Store} \to \text{Env} \times \text{Store}$$

where the value is an *initializer.*

Definition 2.7 *Let $d \in$ Decl be a declaration, and $\mathcal{E}$, $\mathcal{S}$ an environment and a store, respectively. Let $v \in$ Value. Then the evaluation of d yields environment $\mathcal{E}'$ and $\mathcal{S}'$ if*

$$\vdash^{decl} \langle d, v, \mathcal{E}, \mathcal{S} \rangle \Rightarrow \langle \mathcal{E}', \mathcal{S}' \rangle$$

where $\vdash^{decl}$ is defined in Figure 14. It holds that $\mathcal{S}'(\mathcal{E}'(x)) = v$ where x is the declarator of d. □

The rules are justified as follows. Only variable definitions cause memory to be allocated. The semantics of allocation is expressed via the relation $\vdash^{alloc}$ briefly defined below. The rules for external variable and functions simply update the environment with the address of the declarator.

$$\text{[var]}\quad \frac{\vdash^{alloc} \langle T, \mathcal{S}\rangle \Rightarrow \langle l, \mathcal{S}'\rangle}{\vdash^{decl} \langle x:\ T, v, \mathcal{E}, \mathcal{S}\rangle \Rightarrow \langle \mathcal{E}[x \mapsto l], \mathcal{S}'[v/l]\rangle}$$

$$\text{[evar]}\quad \frac{\text{Address of } v \text{ is } l}{\vdash^{decl} \langle \texttt{extern}\ v:\ T, v, \mathcal{E}, \mathcal{S}\rangle \Rightarrow \langle \mathcal{E} \circ [v \mapsto l], \mathcal{S}'\rangle}$$

$$\text{[efun]}\quad \frac{\text{Address of } f \text{ is } l}{\vdash^{decl} \langle \texttt{extern}\ f:\ T, v, \mathcal{E}, \mathcal{S}\rangle \Rightarrow \langle \mathcal{E} \circ [v \mapsto l], \mathcal{S}'\rangle}$$

Figure 14: Dynamic semantics for declarations

2.4.7 Memory management

Precise modeling of storage is immaterial for our purposes, and further depends on an implementation. The Standard requires an implementation to adhere to the following requirements which we take as axioms.

- Variables shall be allocated in sequence. Objects of base type occupy a suitable number of bytes depending on the machine. Members of structs shall be allocated in order of declaration such that member 1 is allocated at location 0 of object. Elements of arrays shall be stored by row.
- The store is managed as a stack.
- Run-time memory allocation is accomplished from a separate space of storage.

We use the relation $\vdash^{alloc}$: Type $\times$ Store $\rightarrow$ Loc $\times$ Store to model the allocation of an object of a particular type. Intuitively, for a type T and store $\mathcal{S}$, $\vdash^{alloc} \langle T, \mathcal{S}\rangle \Rightarrow \langle l, \mathcal{S}'\rangle$ holds if $\mathcal{S}'$ is the store resulting from the allocation in store $\mathcal{S}$ of memory for an object of type T at location l. We ignore the deallocation aspect.

2.4.8 Some requirements

The following properties are of importance for later chapters, and therefore briefly mentioned here.

- A pointer to a struct or union points also, when suitably converted, to the first member [ISO 1990, Paragraph 6.5.2.1].
- Common initial members of members of structs in a union are guaranteed to be shared. Thus, if a struct member of a union is assigned, the value can be accessed via another union member [ISO 1990, Paragraph 6.3.2.3].
- Pointer arithmetic is only allowed to move a pointer around inside an array or one past the last element. Programs may not rely on variables to be allocated in sequence (except members of a struct that are allocated in order of definition, but where padding may be inserted) [ISO 1990, Paragraph 6.3.6].

This completes the description of the dynamic semantics of abstract C. We conjecture that the semantics complies to the Standard. Notice, however, the semantics in some respect is more specified than the Standard. For instance, the evaluation order is fixed. Since conforming programs are not allowed to rely on evaluation order, this does not violate the correctness of the semantics.

2.5 Related work

In our Master's Thesis we gave an operational semantics for Core C, a subset of C including functions, global variables, structs and arrays [Andersen 1991]. Gurevich and Huggins have specified the semantics of the core part of C by the means of *evolving algebras* [Gurevich and Huggins 1992].

Computation of Fortran programs' call graphs has been studied in numerous works [Ryder 1979,Callahan *et al.* 1990,Lakhotia 1993]. Contrary to C, Fortran has procedure variables, but no function pointers. Hendren *et al.* have developed a context-sensitive invocation graph analysis supporting function pointers [Hendren *et al.* 1993]. Their analysis is coupled with a point-to analysis which approximates the set of functions a function pointer may point to. Thus, their invocation graphs are more precise than ours, but more expensive to compute.

The literature reports several program representations, each suitable for different purposes. Hendren *et al.* use a Simple representation where complex expressions are broken down into three-address code [Hendren *et al.* 1992]. The Miprac environment by Harrison *et al.* uses a representation where computation is expressed via primitive operations [Harrison III and Ammarguellat 1992].

2.6 Summary

We have described the abstract syntax and semantics of the C programming language. Furthermore, we have devised a representation that accommodates user feedback from program analysis, and described static-call graphs.

Chapter 3

Generating extensions

We develop an automatic *generating extension generator* for the Ansi C programming language. A generating extension of a program p produces, when applied to partial program input s, a specialized version p_s of p with respect to s. A generating extension generator is a program transformation that converts a program into its generating extension.

Partial evaluation is a quickly evolving program specialization technique that combines generality with efficiency. The technique is now being applied to realistic applications and languages. Traditionally, partial evaluation has been accomplished by partial evaluators '`mix`' based on symbolic evaluation of subject programs. Expressions depending solely on known input are evaluated while code is generated for constructs possibly depending on unknown values.

This approach has several drawbacks that render the development of '`mix`' for realistic imperative languages hard. The conspicuous problems include preservation of semantics, memory usage, and efficiency. To evaluate static expressions, '`mix`' contains an evaluator. However, in the presence of multi-level pointers, user defined structs, an address operator, casts, function pointers and implementation-defined features such as *e.g.* a `sizeof` operator, semantical correctness becomes hard to establish, both in practice and in theory. Further, the memory usage of a specializer is considerably larger than by direct execution, sometimes beyond the acceptable. Finally, symbolic evaluation is known to be an order of magnitude slower than direct execution. In practice this may make specialization infeasible if possible at all.

A generating extension, on the other hand, evaluates static expressions directly via the underlying implementation. Thus, the semantic correctness of the evaluation comes for free; there is no representation overhead of static values, and no symbolic evaluation occurs.

This chapter investigates specialization of all parts of the Ansi C programming language, and develops the corresponding transformations of constructs into their generating equivalents. We study specialization of functions and their interaction with global variables; program point specialization of statements; treatment of pointers and runtime memory allocation; specialization-time splitting of arrays and structs, and various other aspects.

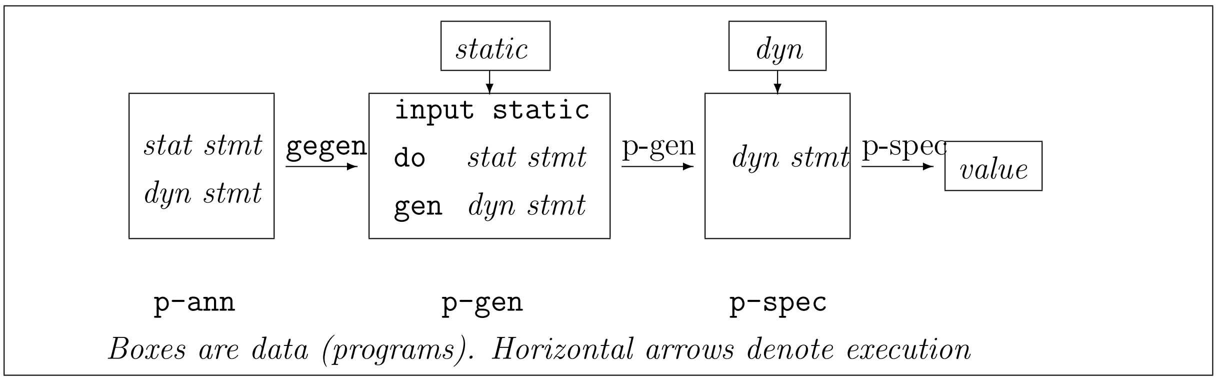

Figure 15: Simplified structure of generating-extension generator

3.1 Introduction

Program specialization by the means of partial evaluation is now a mature and well-understood technology. Even though most research has focussed on functional languages, there has been some progress reported on specialization of imperative programming languages, *e.g.* Fortran [Baier *et al.* 1994], Pascal [Meyer 1992] and C [Andersen 1993b]. During extension of previous work we faced several difficulties that seemed hard to incorporate into existing partial evaluators. Two substantial problems were *preservation* of semantics and *efficiency* — two central issues in automatic program transformation.

An example. The specialization of a ray tracer took several minutes using a previous version of *C-Mix* which was based on symbolic evaluation. The current version, which is based on the *generating extension transformation* developed in this chapter, accomplishes the task in seconds.

What went wrong in the old version of *C-Mix*? The basic problem is that the specializer works on a (tagged) *representation* of the static values and spends much time on tag testing and traversal of the representation. A generating extension, on the other hand, works *directly* on static values.

A simplified view of a generating-extension generator '`gegen`' is shown in Figure 15. Input to the generator is a binding-time annotated program '`p-ann`', and output the generating extension '`p-gen`'. The static parts of the program appear in the generating extension as statements to be executed, whereas the dynamic parts are transformed into code generating statements. When run, the generating extension inputs the static data; executes the static statements, and generates code for the dynamic statements. The output, '`p-spec`', is a specialized version '`p-spec`' of '`p`'. The specialized program inputs the dynamic data and yields a value.

In this chapter we develop a *generating-extension generator* that transforms a program p into its generating extension p_{gen}. Since the transformation depends on the specialization of program constructs, *e.g.* expressions, statements, data structures *etc.*, we also cover specialization of all aspects of the C programming language.

3.1.1 The Mix-equation revisited

Recall the Mix-equation from Section 1.2.1. Let p be a two-input subject program in language C over data domain D, and suppose $s, d \in D$. Execution of p on s, d is denoted by $[\![p]\!]_C(s,d) \Rightarrow v$, where $v \in D$ is the result (assuming any is produced). By $\overline{p}^{pgm}$ we denote the representation, or encoding in C, of p into some data structure *pgm*.

A *partial evaluator* '`mix`' fulfills that

$$\text{if} \quad [\![\texttt{mix}]\!]_C(\overline{p}^{pgm}, \overline{s}^{val}) \Rightarrow \overline{p_s}^{pgm} \quad \text{and} \quad [\![p]\!]_C(s,d) \Rightarrow v \quad \text{then} \quad [\![p_s]\!](d) \Rightarrow v$$

The use of implication "if" as opposed to bi-implication allows '`mix`' to loop.

Traditionally, partial evaluation has been accomplished by the means of symbolic evaluation of subject programs. Operationally speaking, '`mix`' evaluates the static constructs[1] and generates code for the dynamic constructs. To evaluate static expressions and statements, '`mix`' contains an interpreter. Disadvantages of this interpretative approach include:

- The evaluator must correctly implement the semantics of the subject language.
- During the specialization, the subject program must be represented as a *pgm* data structure inside '`mix`'.
- Static data must be encoded into a uniform data type for representation in '`mix`'.[2]
- Symbolic evaluation is an order of magnitude slower than execution.
- Compiler and program generator generation is hindered by double encoding of the subject programs, cf. the Futamura projections in Section 1.2.2.

To evaluate static expressions '`mix`' employs an evaluator which must be faithful to the Standard, *i.e.* a conforming implementation of C. However, implementation of such an evaluator is both a non-trivial and error prone task. Consider for example interpretation of pointer arithmetic; conversion (cast) of a pointer to a struct to a pointer to the first member; representation of function pointers and evaluation of the address operator. Even though it in principle is possible to implement a correct evaluator, it is in practice hard to establish its correctness.

Furthermore, evaluators work on *representations* of values. For example, a value can be represented (assuming the evaluator is written in C) via a huge union Value with entries for integers, doubles, pointers to Values, *etc.* The possibility of user-defined structs and unions renders this representation problematic.

Consider representation of structs. Since the number of members is unknown,[3] struct must be represented external to a Value object, for instance by a pointer to an array of Values representing the members. Now suppose that a call (in the subject program)

[1] An expression is classified 'static' if it solely depends on known input. Otherwise it is 'dynamic'.

[2] In dynamically-typed languages, for example Lisp, the 'encoding' is done by the underlying system.

[3] Actually, the Standard allows an upper limit of 127 members in a struct but this seems rather impractical [ISO 1990, Paragraph 5.2.4.1].

passes a struct as argument to a function, and recall that structs are passed call-by-value. This representation does not, however, "naturally" implement call-by-value passing of structs: when the representation of a struct is copied (call-by-value) only the pointer to the array is copied, not the array. The copying must be done explicitly, necessitating objects to be tagged at mix-time with their size and type. This increases memory usage even further [Andersen 1991,Launchbury 1991,De Niel *et al.* 1991]. Some empirical evidence was given in the introduction; the specializer spends a considerably time comparing tags and traversing the representation of static data.

Finally, consider generation of program generators by self-application of '`mix`'. In previous work we succeeded in self-applying the specializer kernel for a substantial subset of C [Andersen 1991], but the program generators suffered greatly from the inherited representation of programs and values. For example, if a generating extension of the power function '`pow()`' was produced by self-application of the specializer to '`pow()`', the specialized programs obtained by execution of the extension '`pow-gen()`' used the indirect Value representation of objects — this is the difference between '`cogen`' and '`gegen`', see Section 1.2.3. It is highly undesirable when specialized programs are intended to be linked with other C programs.

3.1.2 Generating extensions and specialization

Recall that a generating extension p_{gen} of a program p is a program that produces specialized versions of p when applied to parts s of p's input. Let $s, d \in D$ and assume $[\![p]\!]_C(s, d) \Rightarrow v$. Then one has

$$\text{if} \quad [\![p_{gen}]\!]_C(s) \Rightarrow \overline{p_s}^{pgm} \quad \text{then} \quad [\![p_s]\!](d) \Rightarrow v,$$

where $v \in D$ is the result. Thus, a generating extension, applied to input s, yields a specialized program p_s of p with respect to s.

A *generating-extension generator* '`gegen`' takes a program p and returns a generating extension p_{gen} of p:

$$[\![\texttt{gegen}]\!]_C(\overline{p_{ann}}^{pgm}) \Rightarrow \overline{p_{gen}}^{pgm}$$

cf. Section 1.2.3. Notice that the input to '`gegen`' actually is a *binding-time annotated* program, and not just a program, as Section 1.2.3 suggests.

A transformer '`gegen`' can be employed to specialize programs in a two-stage process:

$$\begin{array}{lll} [\![\texttt{gegen}]\!](\overline{p_{ann}}^{pgm}) & \Rightarrow \overline{p_{gen}}^{pgm} & \text{Produce the generating extension} \\ [\![p_{gen}]\!]_C(s) & \Rightarrow \overline{p_s}^{pgm} & \text{Produce the specialized program} \end{array}$$

for all programs p and static input s.

Operationally speaking, a generating extension generator works as follows. Given a binding time annotated program, it in essence copies static constructs unchanged into p_{gen} for evaluation during execution of the extension. Dynamic constructs in p are transformed into calls in p_{gen} to code generating functions that produce residual code. The residual code makes up the specialized program. This is illustrated in Figure 15.

Both '`gegen`' and the automatically produced compiler generator '`cogen`' produce generating extensions, but the residual programs (may) differ.[4] Since '`cogen`' uses the same representation of objects as '`mix`', specialized programs inherit the Value representation. On the other hand, '`gegen`' uses a direct representation of values, and produces residual declarations with the same types as given by the subject program.

Consider again the issues listed in the previous section.

- During specialization, that is, when the generating extension is run, static constructs are evaluated directly by the underlying implementation, in other words an extension is compiled and run as a ordinary program. Thus there is no need for an interpreter.
- In the generating extension, the subject program is not "present", that is, the extension does not carry around a program data structure. Only the generating extension generator needs the subject program.
- Static values are represented directly. There is no need for an expensive (in time and storage usage) universal Value encoding.
- There is essentially no interpretation overhead: static statements and expressions are evaluated, not interpreted.[5]
- The program generator '`gegen`' does not inherit its types from '`mix`'. Hence, program generators created by '`gegen`' are (usually) more efficient than possible with '`cogen`'.

In addition, we identify the following advantages.

Suppose that a program is to be specialized with respect to several different static values. Using '`mix`', the subject program must be evaluated for every set of static values. By using '`gegen`', the subject program need only be examined once — when the generating extension is produced. The static computations will be redone, but not the source program syntax analysis *etc.*

Next, suppose that a commercial product uses program specialization to speed up program execution. Using the '`mix`' technology, a complete partial evaluator must be delivered with the software; by using '`gegen`', only the generating extension (which is derived from the original software) needs to be released.

Naturally, the same effect can, in principle, be achieved by '`cogen`', but this requires a self-applicable '`mix`', and it does not solve the representation problem.

Finally, due to the simplicity of the generating extension transformation, it becomes possible to argue for the correctness — a task which seems hopeless in the case of a traditional partial evaluator for any non-trivial language.

Definition 3.1 *(The Gegen-equation) Let p be a program in a language C over data domain D, and $s \in D$. A generating extension generator* '`gegen`' *fulfills that*

$$\textit{if}\quad [\![\texttt{gegen}]\!](\overline{p_{ann}}^{pgm}) \Rightarrow \overline{p_{gen}}^{pgm} \quad \textit{then}\quad [\![p_{gen}]\!]_C(s) \Rightarrow \overline{p_s}^{pgm}$$

[4]However, it requires the existence of a self-applicable mix.

[5]A truth with modification: statements and expressions are evaluated directly, but the memory management imposes a penalty on the program execution, see Section 3.10.

where p_s is a residual version of p with respect to s. □

Naturally, a program generator must always terminate. However, we allow a generating extension to suffer from the same imperfect termination properties as specializers. That is, they may fail to terminate on some (static) input s when the ordinary program execution terminates on s and all dynamic input. Nontermination can be due to two reasons. Either the subject program loops on the static data for all dynamic input. It is normally acceptable for a specializer to loop in this case. However, it may also be the case that the generating extension loops even though normal direct execution terminates. This is, of course, annoying. In the following we assume that generating extensions terminate "often" enough. In Section 3.14 we consider the termination problem in greater detail.

3.1.3 Previous work

Ershov introduced the term *generating extension* $G(p)$ of a program p, defined such that $[\![G(p)]\!](x) = [\![M]\!](p, x)$ for all data x, where M is a "mixed computation". The framework was developed for a small toy language with assignments and while-loops.

Pagan described hand-writing of generating extensions for some example programs [Pagan 1990]. The techniques are to our knowledge not extended nor automated to cope with languages supporting pointers, user-defined structs, runtime memory allocation *etc.* in a systematic way.

The RedFun project contained a hand-written compiler generator for a subset of Lisp [Beckman *et al.* 1976]. The system aimed towards optimization of compilers by elimination of intermediate computation rather than specialization of programs.

Holst and Launchbury suggested hand-writing of '`cogen`' mainly as a way to overcome the problem with double-encoding of subject programs at self-application time. They studied a small ML-like language without references. The key observation made was that the structure of an automatically generated compiler resembles that of a hand-written compiler, and it therefore is a manageable task to write a compiler generator by hand.

Birkedal and Welinder have developed a generating extension generator for the Core Standard ML language (without imperative constructs) [Birkedal and Welinder 1993]. Bondorf and Dussart have recently studied a CPS-cogen for an untyped lambda calculus. Their work remains to be generalized to a more realistic context, though.

The author has succeeded in self-application of a partial evaluator for a subset of the C language [Andersen 1991,Andersen 1993b]. We managed to generate '`cogen`', and used it to convert interpreters into compilers, and more broadly, programs into generating extensions. Section 3.15 contains a detailed comparison with related work, and cites relevant literature.

The present work extends previous work as follows. We develop partial evaluation for the full ANSI C language; the present work allows specialization of larger programs than previously possible, and we demonstrate that our techniques are feasible in practice. Furthermore, the aim of the present work is different. Whereas previous work mainly was concerned with self-application and compiler generation, the present work focusses primarily on efficient specialization and its use in software development.

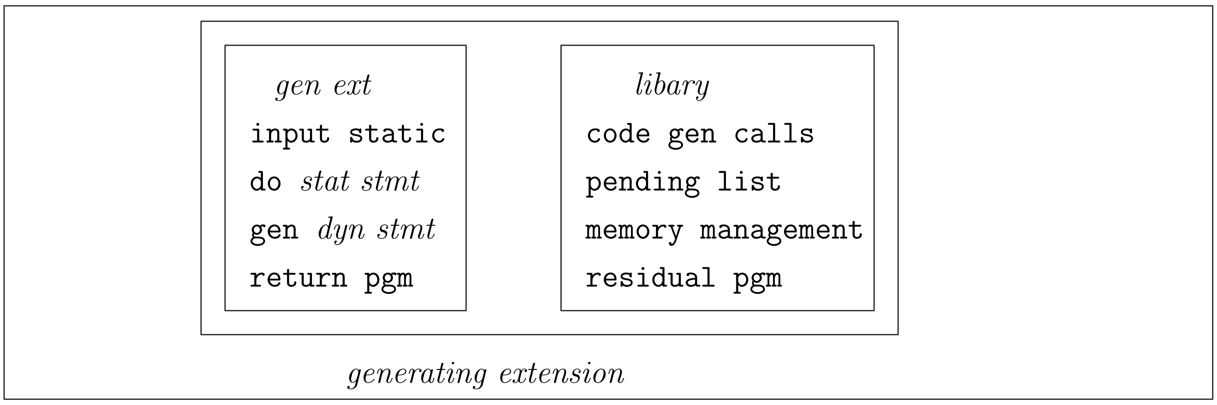

Figure 16: Structure of a generating extension

3.1.4 Overview of chapter

Knowledge of polyvariant program-point specialization and realization of these techniques are assumed throughout the chapter [Gomard and Jones 1991a,Jones *et al.* 1993]. The rest of the chapter is structured as follows.

The following section presents an example: transformation of a string-matcher into its generating extension. The section serves to introduce basic concepts and notions used in the later exposition.

Sections 3.3 to 3.8 contain a complete treatment of all aspects of the C programming language. This part of the chapter is modeled over the book "*the C programming language*" by Kernighan and Ritchie [Kernighan and Ritchie 1988]. We consider specialization of expressions, statements, functions, and the interaction between global variables and side-effects. Further, we consider treatment of pointers, arrays and structs; define the notion of partially-static data structure and specialization-time splitting of these, and runtime memory allocation. These sections serve three purposes: they describe specialization of the various constructs; they introduce the corresponding transformations, and they state the *binding time separation requirements* that a subject program must meet. We do not consider binding-time analysis in this chapter; subject programs are assumed to be annotated with binding times. Chapter 3.9 summarizes the transformation.

Section 3.10 investigates the memory management in a generating extension. Due to pointers and side-effects the memory management is more involved than in for example functional languages. Furthermore, storage usage is of major concern: specialization of big programs may easily exhaust the memory unless care is taken. Section 3.12 describes some strategies for function specialization, sharing and unfolding. Section 3.11 considers code generation, and introduces an improvement that allows determination of (some) dynamic tests.

Section 3.14 focusses on the termination problem. We argue that methods reported in the literature are insufficient for realistic use.

Finally, Section 3.15 describes related work, and Section 3.16 lists related work and concludes.

An refined picture of a generating extension is shown in Figure 16.

```
/* Copyright (C) 1991, 1992 Free Software Foundation, Inc. */
/* Return the first occurrence of NEEDLE in HAYSTACK. */
char *strstr(char *haystack, char *needle)
{
  register const char *const needle_end = strchr(needle,'\0');
  register const char *const haystack_end = strchr(haystack,'\0');
  register const size_t needle_len = needle_end - needle;
  register const size_t needle_last = needle_len - 1;
  register const char *begin;

  if (needle_len == 0) return (char *) haystack; /* ANSI 4.11.5.7 */
  if ((size_t) (haystack_end - haystack) < needle_len) return NULL;
  for (begin = &haystack[needle_last]; begin < haystack_end; ++begin) {
      register const char *n = &needle[needle_last];
      register const char *h = begin;
      do
         if (*h != *n) goto loop; /* continue for loop */
      while (--n >= needle && --h >= haystack);
      return (char *)h;
    loop:;
    }
  return NULL;
}
```

Figure 17: The GNU implementation of standard string function '`strstr()`'

3.2 Case study: A generating string-matcher

The aim of this section is to illustrate the main principles by a case study: transformation of a (naive) string-matcher program into its generating extension. The string-matcher inputs a pattern and a string, and returns a pointer to the first occurrence of the pattern in the string (and `NULL` otherwise). The generating extension takes a pattern, and yields a specialized matcher that inputs a string. As subject program we use '`strstr()`'.

The speedup we obtain is not *very* impressive, but the example is simple enough to convey how a generating extension works. Further, the specialized matchers are not optimal: they contain redundant tests, and do not compare with for example Knuth, Morris and Pratt (KMP) matchers [Knuth *et al.* 1977]. Recently specialization of string matchers has become a popular test for comparing "strength" of program transformers [Jones 1994]. We return to this problem in Chapter 10 where we consider the stronger specialization technique *driving*, that allows propagation of context information. This enables generating of efficient KMP matchers from naive string matchers.

3.2.1 The string-matcher `strstr()`

The GNU C library implementation of the `strstr()` function is shown in Figure 17[6] [Stallman 1991]. The string matcher is "naive" (but efficiently implemented).

[6]This code is copyrighted by the Free Software Foundation.

```
char *strstr_1 (char *haystack)
{
   char *haystack_end;
   char *begin;
   char *h;
   haystack_end = strchr (haystack, '\0');
   if ((int) haystack_end - haystack < 3) return 0;
   begin = &haystack[2];
 cmix_label4:
   if (begin < haystack_end) {
      h = begin;
      if (*h-- != 'b') {
         ++begin;
         goto cmix_label4;
      } else
         if (*h-- != 'a') {
            ++begin;
            goto cmix_label4;
         } else
            if (*h-- != 'a') {
               ++begin;
               goto cmix_label4;
            else
               return (char *) h + 1;
      }
   } else
      return 0;
}
```

Figure 18: Specialized version of '`strstr()`' to `"aab"`

First, the lengths of the pattern '`needle`' and the string '`haystack`' are found via the standard function '`strchr()`'. If the length of the pattern exceeds the length of the string, no match is possible. Otherwise the pattern is searched for by a linear scan of the string in the `for` loop.

Suppose that we examine DNA strings. It is likely that we look for the same pattern among several strings. It may then pay off to specialize '`strstr()`' with respect to a fixed pattern, to obtain a more efficient matcher. In the following, '`needle`' is *static* (known) data, and '`haystack`' is *dynamic* (unknown) input.

The goal is a *specialized* program as shown in Figure 18. The figure shows '`strstr()`' specialized to the pattern '`"aab"`'.[7] Input to the program is the string '`hay_stack`'. The pattern has been "coded" into the control; there is no testing on the '`needle`' array.

3.2.2 Binding time separation

Consider the program text in Figure 17. When '`haystack`' is classified dynamic, all variables depending on it must be classified dynamic as well. This means that '`haystack_end`',

[7]The program was generated automatically, henceforth the "awful" appearance.

```
char *strstr(char *haystack, char *needle)
{
  register const char *const needle_end = strchr(needle, '\0');
  register const char *const haystack_end = strchr(haystack, '\0');
  register const size_t needle_len = needle_end - needle;
  register const size_t needle_last = needle_len - 1;
  register const char *begin;
  if (needle_len == 0) return (char *) haystack; /* ANSI 4.11.5.7 */
  if ((size_t) (haystack_end - haystack) < needle_len) return NULL;
  for (begin = &haystack[needle_last]; begin < haystack_end; ++begin) {
      register const char *n = &needle[needle_last];
      register const char *h = begin;
      do
         if (*h-- != *n) goto loop; /* continue for loop */
      while (--n >= needle);
      return (char *)h+1;
    loop:;
    }
  return NULL;
}
```

Figure 19: Binding-time separated version of '`strstr()`'

'`begin`' and '`h`' become dynamic. On the other hand, '`needle_len`' and '`needle_last`' are static. A separation of variables into static and dynamic is called a *division*. The classification of '`haystack`' and '`needle`' is the *initial division*.

In Figure 19 dynamic variables, expressions and statements are underlined.

A minor change has been made for the sake of presentation: the conjunct '`--h >= haystack`' has been eliminated from the do-while loop. It is easy to see that the modified program computes same function as the original. The change is necessary to prevent the loop from becoming dynamic. In Section 3.11 we describe an extension which renders this binding time improvement unnecessary.

The `for` loop and the `if` statement are dynamic since the tests are dynamic. A function must return its value at runtime, not at specialization time, hence the `return` statements are classified dynamic, including the '`return NULL`'.

3.2.3 The generating extension `strstr-gen()`

The generating extension shall output a specialized version of '`strstr()`' with respect to '`needle`'. Intuitively, this is accomplished as follows.

Suppose that the program in Figure 19 is "executed". Non-underlined (static) constructs are be executed as normally, *e.g.* the do-while loop iterated and '`--n`' evaluated. When an underlined (dynamic) construct is encountered, imagine that the statement is added to the residual program, which is being constructed incrementally. When the "execution" terminates, the residual program will contain specialized versions of all those statements that could not be executed statically. Due to loops, a statement may get specialized several times, *e.g.* the `if` test can be recognized several times in Figure 18.

The type `Code`. To represent dynamic variables we introduce a reserved type '`Code`' in generating extensions. For instance, the definition '`Code haystack`' represents the dynamic pointer '`haystack`'. Variables of type '`Code`' must appear in the residual program. In this case, '`haystack_end`', '`begin`' and '`h`' will appear in specialized versions of '`strstr()`', as expected.

Residual code and code generating functions. During the execution of a generating extension, residual code is generated. We assume a library of *code generating functions* that add residual statements to the residual program. For example, '`cmixExpr(`$\overline{e}$`)`' adds an expression statement $\overline{e}$ to the residual program, and '`cmixIf(`$\overline{e}$`,`m`,`n`)`' adds '`if (`e`) goto` m`; else goto` n'.

Similarly, we assume code generating functions for expressions. For example, the function '`cmixBinary(`$\overline{e_1}$`,"+",`$\overline{e_2}$`)`' returns the representation of a binary addition, where $\overline{e_1}$ and $\overline{e_2}$ are representations of the arguments. The function '`cmixInt()`' converts an integer value to a syntactic constant.

Transformation of non-branching statements. Consider first the expression statement '`begin = &haystack[needle_last]`'. Recall that '`begin`' and '`haystack`' are dynamic while '`needle_last`' is static. The result of the transformation is

```
cmixAssign(begin,"=",cmixAddress(cmixArray(haystack,cmixInt(needle_last))))
```

where '`cmixInt()`' "lifts" the static value of '`needle_last`' to a syntactic constant.

Consider the dynamic statement '`return (char *)h+1`'. The transformed version is:

```
cmixReturn(cmixCast("char *", cmixBinary(h,"+",cmixInt(1)))); goto cmixPendLoop;
```

where '`cmixPendLoop`' is a (not yet specified) label.

First, a residual return statement is added to the residual program by the means of '`cmixReturn()`'. Since a `return` terminates a control-flow path, there is nothing more to specialize. The '`goto`' statement transfers control to the pending loop (see below).

Transformation of branching statements. If the test of an `if` or a loop is static, it can be determined at specialization time. Consider the `do` S `while (--n >= needle)` in the subject program (Figure 19). Since '`n`' and '`needle`' are static, the loop is static, and can be unrolled during specialization. The loop is henceforth transformed into `do` $\overline{S}$ `while (--n >= needle)` in the generating extension, where $\overline{S}$ is the transformation of S.

Consider specialization of the dynamic conditional '`if (*h!=*n) goto loop;`'. Similarly to a compiler that must generate code for both branches of an `if`, we must arrange for both branches to be specialized. Since only one branch can be specialized at a time, we introduce a *pending list* of program points pending to be specialized. Let the library function '`cmixPendinsert()`' insert the "label" of a program point to be specialized.

The conditional is transformed as follows:

`cmixIf(`$\overline{e}$`,cmixPendinsert(`m`), cmixPendinsert(`n`)),` where the m is the label of '`goto loop`' (inserted during the transformation), and n is the label of the (empty) statement after the `if` (more generally, the label of the else branch), and $\overline{e}$ is the transformation of the test.[8] After the then-branch, a jump '`goto end_if`' is inserted, where '`end_if`' is the label of the statement following the `if`.

We implicitly convert a loop with a dynamic test into an `if-goto`.

The pending loop. The *pending loop* in a generating extension is responsible to ensure that all program points inserted into the pending list are specialized. While program points are pending for specialization, one is taken out and marked "processed", and the corresponding statements in p_{gen} are executed.

```
Code strstr_gen(char *needle)
{
   ...
   int cmixPP;
   cmixPendinsert(1);
   while (cmixPP = cmixPending()) {
      cmixLabel(cmixPP);
      switch (cmixPP) {
         case 1: /* Program point 1 */
         ...
         case n: /* Program point n */
         cmixPendLoop:;
         }
    }
   return;
}
```

To initiate the specialization, the label of the first statement is inserted into the list.

We have left out one aspect: a program point shall be specialized with respect to the values the static variables assumed when the label was inserted into the pending list. For now we simply assume that '`cmixPending()`' (somehow) records this information, and restores the values of the static variables.

Example 3.1 Consider the following (contrived) version of sign, and assume that it is specialized with respect to a dynamic value '`x`'.

```
/* sign: return sign of v */
int sign(int x)
{
   int v = 0;
   if (x >= 0) v += 1; else  v -= 1;
   return v;
}
```

Suppose that the then-branch is specialized first. During this, the value of '`v`' will be updated to 1. Eventually, the else-branch is specialized. Before this, however, the static variable '`v`' must be restored to 0, which was its value when the else-branch was recorded for specialization. **End of Example**

[8]In this example we courageously rely on left-to-right evaluation order.

```
Code strstr(char *needle)
{
  register const char *const needle_end = strchr(needle,'\0');
  register const size_t needle_len = needle_end - needle;
  register const size_t needle_last = needle_len - 1;
  Code haystack, haystack_end, begin;
  int cmixPP;
  cmixPendinsert(1);
  while (cmixPP = cmixPending()) {
     cmixLabel(cmixPP);
     switch(cmixPP) {
        case 1: cmixExpr(haystack_end = strchr(haystack, '\0'));
                if (needle_len == 0)
                   { cmixReturn((char *)haystack); goto cmixPendLoop; }
                cmixIf((size_t) (haystack_end - haystack) < needle_len,
                       cmixPendinsert(2),
                       cmixPendinsert(3));
                goto cmixPendLoop;
        case 2: cmixReturn(cmixInt(NULL)); goto cmixPendLoop;
        case 3: cmixExpr(begin = &haystack[needle_last]);
                for_loop: cmixIf(begin < haystack_end;
                                 cmixPendinsert(4),
                                 cmixPendinsert(9));
                goto cmixPendLoop;
        case 4: register char *n = &needle[needle_last];
                cmixExpr(h = begin);
                do {
                   cmixIf(*h-- != cmixLift(*n),
                          cmixPendinsert(8),
                          cmixPendinsert(5)));
                   goto cmixPendLoop;
                   end_if: cmixExpr(--h);
                }
                while (--n >= needle);
                cmixReturn((char *)h+1);
                goto cmixPendLoop;
        case 5: goto end_if;
        case 6: cmixReturn ((char *)h);
                goto cmixPendLoop;
        case 8: goto loop;
        case 7: loop: cmixExpr(begin++);
                goto for_loop;
        case 9: cmixReturn(cmixInt(NULL));
                goto cmixPendLoop;
          cmixPendLoop:;
     }
  }
  return  name_of_function;
}
```

Figure 20: The generating extension of '`strstr`*'*

In practice, static values must be *copied* when a label is inserted into the pending list, and *restored* when the label is taken out via '`cmixPending()`'. A program point and the values of the static variables are called a *partial state*.

The generating extension transformation. The transformation of '`strstr()`' into its generating extension can be summarized as follows.

1. Change the type of dynamic variables to '`Code`'.
2. Copy static expressions and statements to the generating extension.
3. Replace dynamic expressions and statements by calls to code generating functions.
4. Insert the pending loop.

The result, the generating extension of '`strstr()`', is shown in Figure 20. Due to lack of space we have not shown calls that generate code for expressions.[9]

3.2.4 Specializing '`strstr()`'

Execution of the generating extension on the input pattern '`"aab"`' yields the residual program shown in Figure 18.[10] Some benchmarks are provided in the table below. All the experiments were conducted on a Sun SparcStation II with 64 Mbytes memory, compiled using the Gnu C with the '`-O2` option'. Each match was performed 1,000,000 times.

Input		Runtime (sec)		
Pattern	String	Original	Specialized	Speedup
aab	aab	4.2	2.5	1.6
aab	aaaaaaaaab	8.5	5.7	1.4
aab	aaaaaaaaaa	7.9	5.6	1.4
abcabcacab	babcbabcabcaabcabcabcacabc	22.4	15.5	1.4

3.3 Types, operators, and expressions

In this and the following sections we describe binding time separation and transformation of the various aspects of the C programming language. The sections also serve to introduce the code generating functions, that the generating-extension library must implement. In this section we consider simple expressions and base type variables.

The generating extension is a C program manipulating program fragments. We assume a type '`Expr`' suitable for representation of expressions.

[9]The program shown is not as generated by *C-Mix*, but adapted to fit the presentation in this section.
[10]We have by hand "beautified" the program.

3.3.1 Constants

The C language supports three kinds of constants: base type constants, string constants, and enumeration constants. A constant is static. To build a syntactic expression corresponding to a constant value, so-called *lift functions* are employed. A lift function builds the syntactic representation of a value. For example, '`cmixInt(int n)`' returns the representation of the integer constant n, and '`cmixString(char *s)`' returns the representation of the string s. The function '`cmixEnum(char *n)`' returns the representation of the named enumeration constant n.[11]

Example 3.2 Consider the '`pow()`' function below and assume that '`base`' is static and '`x`' is dynamic.

```
int pow(int base, int x)
{
   int p = 1;
   while (base--) p = p * x;
   return p;
}
```

The variable '`pow`' is assigned the value of '`x`', and is therefore dynamic. The constant 1 appears in a dynamic context, and henceforth a lift is needed in the generating extension: '`cmixAssign(p,cmixInt(1))`'. **End of Example**

To avoid code duplication, the use of '`cmixString()`' is restricted to string constants. That is, arbitrary character arrays cannot be lifted.

3.3.2 Data types and declarations

We impose a uniform (program-point insensitive) monovariant binding-time division on variables. Variables with definitely specialization-time known values are called *static*, others are called *dynamic*. In later sections we introduce *partially-static* variables.

During specialization, a static variable holds a value and a dynamic variable holds a *symbolic address*. In the generating extension, static variables are defined as in the subject program, and dynamic variables are defined to have the reserved type '`Code`'. In Section 3.6 we argue why it is convenient to let dynamic variables contain their symbolic addresses in the residual program.[12] A variable's symbolic address corresponds to its residual name.

Example 3.3 Consider again the '`pow()`' function defined in Example 3.2. In the generating extension, the static parameter is defined by '`int base`', and the dynamic variables by '`Code x`' and '`Code p`', respectively.[13] **End of Example**

[11]Currently, enumerations are implicitly converted to integers when needed. However, the revision of the Standard is likely require strong typing for enumeration constants.

[12]The obvious alternative would be to have a function such as 'cmixVar("x")' to generate a residual variable reference.

[13]Dynamic variables must also be initialized, *i.e.* Code p = cmixVar("p").

The functions '`cmixPar(char *type,Expr var)`' and '`cmixLocal(char *type,Expr var)`' produce residual parameters and variable definitions.

Example 3.4 The two calls: '`cmixPar("int",x)`' and '`cmixLocal("int",p)`' define the dynamic variables in the residual program of '`pow()`'. **End of Example**

The type of a dynamic variable can be deduced from the subject program.

3.3.3 Operators

The set of operators can be divided into four groups: unary, binary, assignment, and special operators. We consider each group in turn below.

The set of unary operators includes `+`, `-`, and `!`. A unary operator application can be performed at specialization time provided the argument is static. Otherwise the application must be suspended. The library function '`cmixUnary(char *op,Expr arg)`' builds a residual application. The representation of a binary operator application is returned by '`cmixBinary(Expr e1,char *op,Expr e2)`'. A binary operator can be applied at specialization time if both arguments are static. Otherwise it must be deferred to runtime.

Example 3.5 The call '`cmixUnary("-",cmixInt(1))` produces the representation of '`-1`'. The expression '`p * x`' is transformed into '`cmixBinary(p,"*",x)`', cf. Example 3.2. **End of Example**

When a dynamic operator application is transformed, it must be checked whether one of the argument is static. In that case it must be surrounded by a lift function.

Assignment operators are considered below. The pointer dereference operator and the address operator are treated in Section 3.6.

3.3.4 Type conversions

The type of a value can be converted implicitly or explicitly. An implicit conversion happens, for example, when an integer value is passed as argument to a function expecting a `double`. Since generating extensions are executed as ordinary programs and static values are represented directly, no special action needs to be taken.

Explicit conversions, normally referred to as *casts*, change the type of an object to a specified type. For example, '`(char *)NULL`' changes the type of the constant 0 to a pointer. Casts are divided into the categories *portable* and *non-portable*. The cast of 0 to a pointer is a portable cast [ISO 1990, page 53]. The effect of non-portable casts is either implementation-defined or undefined.

A cast between base types can be evaluated at specialization time provided the operand is static. Thus, static casts are transformed unchanged into the generating extension.

Consider the following cases [ISO 1990, Paragraph 6.3.4]:

- Conversion of a pointer to an integral type:[14] The conversion must be suspended since an integral value can be lifted, and it when would become possible to lift specialization-time addresses to runtime.

[14]The effect of the conversion is implementation-defined or undefined.

- Conversion of an integer to a pointer type:[15] This conversion must be suspended since specialized programs otherwise could depend on the implementation that generated the program, *i.e.* the computer that executed the generating extension.
- Conversion of a pointer of one type to a pointer of another type.[16] Since alignments in general are implementation-defined, the conversion must be suspended to runtime.
- Conversion of function pointers.[17] The cast can be performed at specialization time provided the pointer is static.

Residual casts are build via '`cmixCast(char *type,Expr e)`'.

Example 3.6 Given the definitions '`int x, *p`'. The cast in '`x = (int)p`' must be suspended, since a pointer is converted to an integral value. The cast in '`p = (int *)x`' must be suspended, since an integral value is converted to a pointer. The call '`cmixCast("char *",cmixInt(0))`' casts 0 to a character pointer. **End of Example**

3.3.5 Assignment operators

The set of assignment operators includes '`=`' and the compound assignments, *e.g.* '`+=`'. An assignment can be performed during specialization if both the arguments are static. Otherwise it must be suspended. Possibly, a lift function must be applied to a static right hand side subexpression. The library function '`cmixAssign(Expr e1,char *op,Expr e2)`' generates a residual assignment.

Example 3.7 The call '`cmixAssign(p,"*=",x)`' produces a residual assignment, cf. Example 3.2. **End of Example**

The C language supports assignment of struct values, for example, given the definitions

```
struct { int x, y; } s, t;
```

the assignment '`s = t`' simultaneously assigns the fields '`x`' and '`y`' of '`s`' the value of '`t`'. Suppose that the member '`x`' is static and member '`y`' is dynamic. An assignment between objects with mixed binding times is called a *partially-static assignment*. Ideally, the assignment between the '`x`' members should be performed at specialization time, and code generated for the '`y`' member.

In the generating extension, however, an assignment cannot both be evaluated (static) and cause code to be generated (dynamic). To solve the conflict, the struct assignment can be split into a static and a dynamic part.

[15]The result is implementation-defined.

[16]The result depends on the object to which the pointer points. A pointer to an object of a given alignment may be converted to an object of the same alignment and back, and the result shall compare equal to the original pointer.

[17]A pointer of function type may be converted to another function pointer and back again, and the result shall compare equal to the original pointer.

Example 3.8 Let the definitions '`struct S { int x, a[10], y; } s, t`' be given, and assume that the members '`x`' and '`a`' are static while '`y`' is dynamic. The assignment '`s = t`' can then be transformed into the three lines:

```
s.x = s.y;
memcpy(s.a, t.a, sizeof(s.a));
cmixAssign(s.y, t.y);
```

where the '`memcpy()`' function copies the entries of the array.[18] **End of Example**

The splitting of struct assignments into separate assignments potentially represents an execution overhead since a compiler may generate an efficient struct assignment, *e.g.* via block copy. At least in the case of the Gnu C compiler on the Suns, this is not the case.[19] Our thesis is that the gain will outweigh the loss.

3.3.6 Conditional expressions

For simplicity conditional expressions 'e_1? e_2 : e_3' are transformed into conditional statements, cf. Chapter 2. This way, evaluation of irreconcilable expressions is avoided, *e.g.* if e_2 and e_3 contain assignments.

3.3.7 Precedence and order of evaluation

The standard specifies an evaluation order for some operators, *e.g.* the 'comma' operator. Since a generating extension is executed in a conforming implementation, precedence rules are automatically obeyed.

Execution of a generating extension on a particular implementation fixes the order of evaluation. Since conforming programs are not allowed to rely on evaluation order, this represents no problem, though.

Example 3.9 The evaluation order in the assignment '`a[i] = i++`' is not specified, that is, it is not defined whether '`i`' will be incremented before or after the index is evaluated. When the generating extension and the specialized program is executed on the same platform as the original program, the behavior will be the same. **End of Example**

3.3.8 Expression statements

An expression can act as a statement. If the expression is static, it is transferred unchanged to the generating extension, for evaluation during specialization. Code for dynamic expressions must be added to the residual program under construction. For this, the function '`cmixExpr(Expr)`' is used. It takes the representation of an expression and adds an expression statement to the residual function. That is, the transformation is

$$e; \quad \overset{gegen}{\Longrightarrow} \quad \texttt{cmixExpr}(\overline{e});$$

where $\overline{e}$ is the transformation of e.

[18]The (impossible) assignment 's.a = t.a' would change the semantics of the program.

[19]At the time of writing, we have not implemented partially-static assignments in the *C-Mix* system.

3.4 Control flow

We employ a variant of depth-first *polyvariant program-point specialization* for specialization of statements and control flow [Jones *et al.* 1993]. This section first presents the so-called *pending loop*, which controls the specialization of program points to static values, and next the generating-extension transformation of the various statement kinds.

We assume that all local variable definitions have function scope, cf. Chapter 2. This simplifies the memory management in a generating extension. Consider for example jumps into a block with dynamic variables. Thus, only two scopes exist: global scope and function scope. Due to pointers, objects lexically out of scope may be accessible.

The set of static (visible) objects at a program point is called the *static store*. During execution of a generating extension, several static stores may exist in parallel. For example, since both branches of a dynamic `if` must be specialized, the generating extension must execute both the then- and the else branch. The else-branch must be specialized to the (static) values valid when the test expression was reduced — not the values resulting from specialization of the then branch (assuming the then branch is specialized before the else branch), so the values must be *copied*.

The static store in which computations are carried on, is called the *active store*.

3.4.1 The pending loop

A generating extension uses a *pending list* to represent the set of program points remaining to be specialized with respect to a static store. The list contains quadruples of the form $\langle l, S, l', f \rangle$, where l is the label of the program point to be specialized (*i.e.* a program label), S is a static store (*i.e.* a "copy" of the values of the static variables), and l' is the label of the residual program point (*i.e.* generated when the label was inserted into the list). The flag f indicates whether an entry has been processed.

The following operations on the pending list are needed: insertion, test for exhaustion of pending program points, and a way of restoring the values of static variables from a static store.

- The function '`cmixPendinsert(Label l)`' inserts label l into the pending list. As a side-effect it makes a copy of the active store. If the label and a "similar" static store already exist in the pending list, the residual label associated with that entry is returned, and the program point is not inserted. This implements sharing of residual code. Copying of the active store is the subject of Section 3.10.

- The function '`cmixPending()`' checks whether any program points remain to be processed, and selects one in the affirmative case. Otherwise it returns 0.

- The function '`cmixPendRestore(Label l)`' restores the static store associated with the selected entry and, marks it "processed". The (text of the) residual label is returned.

For simplicity we shall assume that the address of a label can be computed by '`&&l`',

and that “computed goto” is available, *e.g.* ‘`goto *l`’.[20] The pending loop can easily be expressed via a `switch` statement as in Section 3.2, if desired. The pending loop in a generating extension is sketched as Algorithm 3.1.

Algorithm 3.1 *The pending loop.*

```
  cmixPendinsert(&&first_label);
cmixPendLoop:
  if (pp = cmixPending()) {          /* Process next program point */
     npp = cmixPendRestore(pp);      /* Restore state            */
     cmixLabel(npp);                 /* Generate residual label */
     goto *pp;                       /* Jump to program point */
  }
  else /* Nothing more to do */
     return  fun_name;                /* Return name             */
  /* Generating extension for body */
first_label:
```

The label ‘`first_label`*’ is assumed to be unique (introduced during transformation).* □

While any un-processed program points remain in the list, one is selected. The address of the corresponding program point is assigned to the variable ‘`pp`’. Next, the active store is restored with respect to the static store associated with the selected entry. A residual label is added to the residual program by the means of the ‘`cmixLabel(Label)`’ call. Finally, control is transferred to the program point to be specialized, via ‘`goto *pp`’. Assuming `pp` $= \langle l, S, l.', f\rangle$, the effect is to branch to the part of p_{gen} that will generate code for the part of p beginning at statement labeled l.

When all program points have been processed, the name of the (now completed) residual function is returned.

3.4.2 If-else

The transformation of an `if` statement is determined by the binding time of the test. If the test is static, the branch can be performed at specialization time. A static `if` is transformed into an `if` in the generating extension. Otherwise a residual `if` statement must be added to the residual program, and both branches specialized.

Let the statement following an `if` be S_3, and consider ‘`if (`e`)` S_1 `else` S_2`;` S_3’. What should the residual code of these statements look like? Two possibilities exist: S_3 can be *unfolded* into the branches, or the control flow can be *joined* at the specialized version of S_3.

Example 3.10 Two residual versions of ‘`if (`e`)` S_1 `else` S_2`;` S_3’.

```
/* unfolding */            /* joining */
if (e)                     if (e)
   S1'; S3';                  S1';
else                       else
```

[20]These operations are supported by the Gnu C compiler.

S_2'; S_3''; S_2';
                                   S_3';

To the left, S_3 has been unfolded into the branches (and specialized), to the right, the control flow joins at (the specialized version of) S_3. **End of Example**

The former gives better exploitation of static values, but the latter gives less residual code. Suppose, however, that S_1 and S_2 contain conflicting assignments, *e.g.* S_1 assigns -1 to 'x' and S_2 assigns 1 to 'x'. This implies that the control flow cannot be jointed at a single statement.

On-line partial evaluators can side-step the problem by suspending conflicting variables by insertion of so-called explicators for disagreeing variables. An explicator *lifts* a variable to runtime [Barzdin 1988,Meyer 1991]. This technique is unsuitable for off-line partial evaluation, though, since the set of variables to be lifted cannot be determined at binding-time analysis time.

We use a compromise where S_3 is shared if the static stores resulting from specialization of S_1 and S_2 are "similar". The transformation of a dynamic `if` is illustrated by the following example.

Example 3.11 The following program fragment is from a binary-search procedure.

```
if (x < v[mid]) high = mid - 1;
else if (x > v[mid]) low = mid + 1;
else return mid;
```

Suppose that 'v' is static but 'x' is dynamic, and hence the `if` must be specialized. The transformed code is as follows.

```
    cmix_test = x < v[mid];
    cmixIf(cmix_test,cmixPendinsert(&&l0),cmixPendinsert(&&l1));
    goto cmixPendLoop;
l0: /* first then branch */
    high = mid - 1;
    cmixGoto(cmixPendinsert(&&l2)); goto cmixPendLoop;
l1: /* first else branch */
    cmix_test = x > v[mid];
    cmixIf(cmix_test,cmixPendinsert(&&l3),cmixPendinsert(&&l4));
    goto cmixPendLoop;
l3: /* second then branch */
    low = mid + 1;
    cmixGoto(cmixPendinsert(&&l5)); goto cmixPendLoop;
l4: /* second else branch */
    cmixReturn(cmixLift(mid));
    goto cmixPendLoop;
l2: l5:
```

The variable '`cmix_test`' is assigned the if-expression to ensure that all side-effects have completed before the active store is copied via '`cmixPendinsert()`'.[21]

When executed, code of the form

[21]Recall that the evaluation order of arguments is unspecified.

```
if (x < 5)
   if (x > 5)
      ...
   else return 2;
```

is generated. **End of Example**

The transformation is recapitulated below.

```
                           cmix_test = $\overline{e}$;
if ($e$)                   cmixIf(cmix_test,cmixPendinsert(&&m),cmixPendinsert(&&n));
                           goto cmixPendLoop;
  $S_m$;     $\overset{gegen}{\Longrightarrow}$        m: $\overline{S_1}$;
                           cmixGoto(cmixPendinsert(&&l));
else                       goto cmixPendLoop;
  $S_n$;                  n: $\overline{S_2}$;
                           cmixGoto(cmixPendinsert(&&l));
                           goto cmixPendLoop;
                         l:;
```

The statement following the '`if`' is made a specialization point (l). Hence, it will be shared if the static stores valid at the end of the two branches agree.

The function '`cmixIf(Expr e,Label m,Label n)`' adds a residual statement '`if (`e`) goto` m`; else goto` n' to the current residual function. The function '`cmixGoto(Label l)`' adds a residual goto.

3.4.3 Switch

The treatment of a `switch` statement depends on the expression. A static `switch` can be executed during specialization, and is thus copied into the generating extension. In the case of a dynamic `switch`, all `case` entries must be specialized. The transformation is shown below.

```
                                 cmix_test = $\overline{e}$;
switch ($e$) {                   cmixIf(cmixBinary(cmix_test,"==",$v\_1$,
   case $v\_1$: $S_1$                         cmixPendinsert(&&l1),
                                        cmixPendinsert(&&l))));
                                 goto cmixPendLoop;
   case $v\_2$: $S_2$     $\overset{gegen}{\Longrightarrow}$    l:cmixIf(cmixBinary(cmix_test,"==",$v\_2$,
                                        cmixPendinsert(&&l2),
                                        cmixPendinsert(&&m));
                                 goto cmixPendLoop;
   ...                          m: ...
   default: $S_n$                  n:cmixGoto(&&ln);
                                 goto cmixPendLoop;
                                l1: $\overline{S_1}$;
                                l2: $\overline{S_2}$;
                                    ...
}                               ln: $\overline{S_n}$;
```

Observe that the residual code generated due to the above transformation correctly implements cases that "fall through".[22]

Example 3.12 Dynamic `switch` statements are candidate for "The Trick" transformation: statically bounded variation [Jones *et al.* 1993]. The point is that in a branch `case` v_i: S_i the switch expression is known to assume the value v_i, even though it is dynamic. In the terminology of driving, this is called *positive context propagation* [Glück and Klimov 1993]. We consider this in Chapter 10. **End of Example**

3.4.4 Loops — while, do and for

Static loops can be iterated (unrolled) during specialization. Consider dynamic loops, *i.e.* loops with a dynamic test.

A `for` loop corresponds to a `while` loop with an initialization and step expression [Kernighan and Ritchie 1988, Reference manual]. A `do` loop is similar to `while`. It suffices to consider `while` loops. The transformation is given below.

```
                             l: /* the loop test */
                                if_test = e;
while (e)                       cmixIf(if_test,
                                       cmixPendinsert(&&m),
                  gegen                cmixPendinsert(&&n));
                   ==>
                                goto cmixPendList;
   S;                        m: S;
                                cmixGoto(cmixPendinsert(l)); /* iterate */
                                goto cmixPendLoop;
                             n: /* end of loop */
```

Notice that the back-jump is specialized, (potentially) giving better sharing of residual code.

Example 3.13 A tempting idea is to introduce a residual `while`. However, this requires all assignments in a while-body to be suspended, which is excessively conservative in many cases. Structured loops can be re-introduced into residual programs by post-processing. [Ammerguellat 1992,Baker 1977]. **End of Example**

Example 3.14 Below a version of the standard library function '`strcmp()`'[23] is shown. We specialize it with respect to '`s = "A"`' and dynamic '`t`'.

```
/* strcmp: return < 0 if s < t,0 if s==t,> 0 if s > t */
int strcmp(char *s, char *t)
{
   for (; *s == *t; s++, t++)
      if (*s == '\0') return 0;
   return *s - *t;
}
```

[22]We assume that 'break' and 'continue' statements are pre-transformed into goto.

[23]Kernighan & Ritchie page 106 [Kernighan and Ritchie 1988].

```
Code strcmp(char *s)
{
   /* pending loop omitted */
l: if_test = cmixBinary(cmixChar(*s), "==", cmixIndr(t));
   cmixIf(cmixPendinsert(&&m),cmixPendinsert(&&n));
   goto cmixPendLoop;
m: if (*s == '\0')
   { cmixReturn (cmixInt(0)); goto cmixPendLoop; }
   cmixExpr(cmixPost(t, "--"));
   cmixGoto(cmixPendinsert(&&l); goto cmixPendLoop;
n: cmixReturn(cmixBinary(cmixChar(*s), "-", cmixIndr(t)));
   goto cmixPendLoop;
}
```

Figure 21: Generating extension for '`strcmp()`*'*

The generating extension is shown in Figure 21, and a residual program below, for '`s = "a"`'.

```
int strcmp_s(char *t)
{
   if ('a' == *t)
      { t++; if ('\0) == *t) return 0; else return '\0' - *t; }
   else return 'A' - *t;
}
```

Transformation of `while` into '`cmixWhile(`$\overline{e}$`,`$\overline{S}$`)`' would result in monovariant specialization, and hence that the assignment '`s++`' would have to be suspended. **End of Example**

It is possible to improve the sharing of residual code slightly. Observe that a program point only is specialized due to dynamic jumps, *e.g.* from a dynamic `if`. In general, the `if` statement controlling a dynamic loop is rarely the target of a jump *the first* time the "loop" is met. Thus, the label of the `if` will first be inserted into the pending list due to the back-jump. This means the `if` will appear at least twice in the residual program. This can be alleviated by forcing it to be a specialization point:

```
   cmixGoto(cmixPendinsert(&&l));
   goto cmixPendLoop;
l: if_test = e;
   cmixIf(if_test,cmixPendinsert(&&m),cmixPendinsert(&&n));
   goto cmixPendLoop;
m: ...
   goto cmixPendLoop; n:
```

3.4.5 Goto and labels

The C language does not exhibit computed gotos so the target of a jump is always statically known. This implies a goto always can be performed during specialization (to the labelled statement in the generating extension corresponding to the target of the jump in the subject program). Static execution of jumps without generating code is known as *transition compression.* Notice that transition compression only will cause non-termination when the program contains a purely static loop.

It is convenient to specialize some gotos, *e.g.* in connection with loops. The library function '`cmixGoto(Label)`' generates a residual `goto`. The accompanying transformation:

$$\texttt{goto m;} \quad \overset{gegen}{\Longrightarrow} \quad \begin{array}{l}\texttt{cmixGoto(cmixPendinsert(\&\&m));}\\ \texttt{goto cmixPendLoop;}\end{array}$$

as seen many times before in the proceeding examples.

Example 3.15 Copying and restoring of static values may be expensive. In the case of a dynamic goto, there is no need to restore the active store if specialization proceeds *immediately* at the target statement, unless some residual code can be shared. The transformation is changed into

$$\texttt{goto m;} \quad \overset{gegen}{\Longrightarrow} \quad \begin{array}{l}\texttt{cmixGoto(cmixPendinsert(\&\&m));}\\ \texttt{if (!cmixPendfound()) goto m;}\\ \texttt{else goto cmixPendLoop;}\end{array}$$

where '`cmixPendfound()`' returns true if the previous call to '`cmixPendinsert()`' shared a residual program point. **End of Example**

3.5 Functions and program structure

When some arguments s to a function f are known, a specialized — and hopefully optimized — version can be produced. This is the aim of function specialization. The subject of this section is transformation of functions into generating functions. A generating function for f adds a specialized version f_s to the residual program, and returns the name of it. Furthermore, we study the impact of side-effects on residual function sharing and unfolding. In the last part of the section we discuss the '`extern`', '`static`' and '`register`' specifiers, and recursion.

3.5.1 Basics of function specialization

Let $v_1, \ldots, v_m$ be static arguments to f. A generating function f_{gen} for f fulfills that if $[\![f_{gen}]\!](v_1, \ldots, v_m) \Rightarrow \overline{f_{spec}}^{pgm}$ then $[\![f_{spec}]\!](v_{m+1}, \ldots, v_n) \Rightarrow v$, whenever $[\![f]\!](v_1, \ldots, v_n) \Rightarrow v$ for all $v_{m+1}, \ldots, v_n$. Besides parameters, functions may also use and modify non-local variables.

Most C programs do not exhibit extensive use of global variables as other imperative languages but, for instance, a global array is not rare. Restricting function specialization to parameters only is likely to give poor results.

Constraint 3.1 Function specialization shall be with respect to both static parameters and static global variables.

Runtime heap-allocated objects can be seen as global (anonymous) data objects.

Example 3.16 The aim is to specialize function '`S_x()`' below.

```
struct S { int x; } *p;
int main(void) { p=(struct S*)malloc(sizeof(struct S)); return S_x(); }
int S_x(void)  { return p->x; }
```

Clearly, '`S_x()`' must be specialized with respect to both global variable '`p`' and the indirection '`p->x`'. **End of Example**

Suppose that a generating function is called twice (during execution of the generating extension) with the "same" arguments. Seemingly, it is then possible to *share* the specialized function (generated due to the first call), and simply let the second invocation do nothing but return the name of that function. For now we assume a predicate '`seenB4(FunId)`' returns true if a call can be shared. Section 3.12 outlines a strategy for function sharing. In the context of fold/unfold transformations, sharing is known as function *folding* [Burstall and Darlington 1977].

The structure of a generating function is given below.

Algorithm 3.2 *Generating function for f*

```
Code fun( static arguments )
{
   if (seenB4( f ))
      return FunName( f ); /* Share */
   /* Otherwise specialize */
   push_fun( f);            /* New res. fun. */
   /* Pending loop */       /* Specialize body of f */
      /* Nothing more to do */
      return pop_fun();
   /* Generating statements */
   ...
}
```

See also Algorithm 3.1. □

First, the generating function checks whether it has been invoked in a similar context before. In the affirmative case, the name of the shared function is returned immediately to the caller, and no new residual function is constructed. Otherwise a residual function is constructed.

Residual functions are generated in parallel to calls in the generating extension. When a generating function is called, a new residual function is built, and code generation resumes in the previous residual function when the generating function returns. We implement this by a *residual function stack*. When a new residual function is to be constructed, its name is pushed onto the stack by '`push_fun(`*f*`)`'. When the specialization has completed, it is popped by the means of '`pop_fun()`'. The code generating functions, *e.g.* '`cmixExpr()`' add code to the residual function currently on top of the residual function stack.

3.5.2 Functions and side-effects

The following program implements a stack and the '`push()`' function.

```
int stack[MAX_STACK], sp = -1;
/* push: push val on top of stack */
void push(int val)
{
   if (sp < MAX_STACK)
      stack[++sp] = val;
   else
      fprintf(stderr, "Stack overflow\n");
}
```

Suppose that the contents of the stack is dynamic, but the stack pointer is static. By specialization of '`push()`' with respect to a particular value of '`sp`', a function that inserts a value at a specific location in the stack is produced. For example, the version of '`push()`' specialized with respect to '`sp = 3`', assigns its argument to '`stack[4]`'. Notice that after the specialization, the (static) value of '`sp`' is 4.

Suppose a second call to '`push()`' with '`sp = 3`'. Apparently, the call can be shared since the call signature matches the previous invocation. Doing this prevents the value of '`sp`' to be updated — which is wrong (unless '`sp`' is dead).

The extreme solutions are to prohibit static side-effects or to disallow sharing of functions accomplishing static side-effects. The former is likely to degenerate specialization; for instance, initialization of global data structures become suspended. The latter increases the risk for non-termination. Section 3.12 presents a sharing strategy.

Example 3.17 A function with a call '`malloc()`' performs a side-effect on the heap, and can henceforth not be shared. **End of Example**

Consider now the following contrived function which performs *static side-effect under dynamic control.*

```
/* pop_zero: pop if top element is zero */
void pop_zero(void)
{
   if (!stack[sp]) sp--;
   return;
}
```

Since the test is dynamic, both branches must be specialized. One branch ends up in a state where '`sp`' is 2, the other leaves '`sp`' unchanged.

```
/* pop_zero_3: pop if stack[3] is zero */
void pop_zero_3(void)
{
   if (!stack[3]) /* sp == 2 */ return;
   /* sp == 3 */ return;
}
```

The problem is that after the call to the generating function for '`pop_zero()`' , the value of the static variable '`sp`' is unknown (even though it must assume either 2 or 3). Its value is first determined at runtime. However, during specialization, code generation resumes from one point: at the statement following the call. Unlike the handling of dynamic `if`, it is not feasible to "unfold" the subsequent statements "into" the branches.[24] Naturally, by unfolding of the side-effect function, the side-effect can be handled as in the case of local side-effects.

Constraint 3.2 Side-effects under dynamic control shall be suspended.

Chapter 6 develops a *side-effect analysis* to detect side-effects under dynamic control.

3.5.3 Recursion and unfolding

Function specialization must proceed depth-first to preserve the execution order. Functions may be recursive, so a call must be recorded as "seen" before the body is specialized, cf. Algorithm 3.2. This does not guarantee termination. Termination properties are discussed in Section 3.14.

Partial evaluation tends to produce many small functions which obviously should be unfolded (inlined) into the caller. Even though most modern compiler can perform inlining, it is desirable to do the inlining explicitly in the residual program. This often enhances a compiler's opportunity to perform optimizations [Davidson and Holler 1992]. A generating function of a function to be unfolded is shortly called an unfold-able generating function.[25]

Unfolding can either be done during specialization or as a postprocess. This section solely describes unfolding during specialization, which can be accomplished as follows.

1. During execution of the generating extension, before a call to an unfold-able function is made, residual code to assign dynamic actuals to the formal of the called function is made.

2. In generating functions for unfold-able functions, the calls to '`push_fun()`' and '`pop_fun()`' are omitted, such that at specialization time, residual code is added to the callee's residual function.

3. Generating functions for an unfold-able function generate an assignment to a "result" variable and a jump to an "end" label, instead of a residual return.

The above procedure implements function unfolding on a function basis. Alternatively, unfolding can be decided on a call-site basis. Since the structure of a generating function depends heavily on whether it is unfold-able or not, we will not pursue this.

Example 3.18 Suppose function '`pow()`' is unfolded into '`main()`'.

[24]In previous work we have described management of end-configuration for updating of the state when sharing functions with side effect under dynamic control [Andersen 1991]. Experiments have revealed that this is not a fruitful approach so we abandon it.

[25]Notice that it is the residual function that is unfolded — not the generating function!

```
int main(void)
{
   int exp, result; /* dynamic */
   result = pow(2,exp);
   return result;
}
```

The residual function has the following appearance.

```
int main(void)
{
   int exp, result;
   int x;
   {
      x = exp;
      p = 1;
      p *= x;
      p *= x;
      p *= x;
      result = p;
   }
  return result;
}
```

End of Example

Copy propagation can further optimize the code, but this can be done by most optimizing compilers.

3.5.4 External variables

Most C programs consist of several translation units each implementing different parts of the system. A module can refer to a global identifier (variable or function) defined in other modules by the means of '`extern`' declarations.[26]

References to externally defined variables must be suspended to runtime. Further, since externally defined functions may side-effect variables in other modules, such calls must also be suspended.

Constraint 3.3 References to externally defined identifiers shall be suspended.

In practice, this requirement is too strict. For example, mathematical functions such as '`pow()`' are externally declared in `<math.h>`, and thus, not even purely static calls will be computed at specialization time. In Section 3.8 we return to this, and introduce the notion of *pure* function. Pure function calls can be evaluated statically.

Program specialization in the context of modules is the subject of Chapter 7. The chapter also discusses the more dubious treatment of global variables. In C, a global identifier is exported to other modules unless it is explicitly declared local by the means of the `static` specifier. Thus, when looking at a module in isolation, *all* global variables, in principle, have to be suspended. A dramatic consequence: no function specialization will occur. For now we continue to assume that a translation unit makes up the (relevant part of the) whole program, and suspend only externally defined identifiers.

[26]Most C compilers allow implicit declaration of functions returning integer values. We assume that all external references are explicitly declared.

3.5.5 Static variables

The storage specifier `static` has two meanings. When applied to a global identifier, the identifier is given file scope. This has no impact on program specialization and can thus be ignored — at least until Chapter 7, where we consider separate program specialization of modules.

When a local variable is defined `static`, it is allocated statically, and lives between function invocations. As argued in Chapter 2, static identifiers can be treated as unique global identifiers.

3.5.6 Register variable

The `register` specifier is a hint to the compiler, and does not influence program specialization. The storage specifier is copied to the residual program.

3.5.7 Initialization

C allows initialization values to be provided as part of definitions. To simplyfy matters, we rely on a pre-transformation that converts initializations into ordinary assignments.[27]

3.6 Pointers and arrays

The subject of this section is specialization of pointers and arrays. In early partial evaluators, data structures were classified either completely static or completely dynamic. This resulted often in the need for manual binding-time engineering of programs, *e.g.* the splitting of a list of pairs into two lists, to achieve good results. In this section we describe binding-time classification and specialization of partially-static data structures, and the interaction between functions and pointers. The next section deals with structures and unions.

3.6.1 Pointers and addresses

A pointer is a variable containing an address or the constant `NULL`. Classify a pointer *static* if its value and contents definitely are known at specialization time. Otherwise classify it *dynamic*. Static pointers can be dereferenced during specialization; dynamic pointers cannot.

Example 3.19 Consider '`int x,y,a[2],*ip`' and assume that '`x`' is static and '`y`' is dynamic. Pointer '`ip`' is a static pointer to a static object in the assignment '`ip = &x`'. It is a static pointer to a dynamic object in '`ip = &y`'. If e is a dynamic expression, the assignment '`ip = &a[`e`]`' forces '`ip`' to be a dynamic pointer. **End of Example**

[27]There is no special reason for doing this, except that it simplifies the presentation.

A static pointer to a dynamic object is a *partially-static data structure*. When referring to static pointers to static objects it is explicitly written.

Recall that a dynamic variable is bound to its *symbolic runtime address* during specialization. For example, if '`int x`' is a dynamic variable, it would be defined by '`Code x`' in the generating extension, and be bound to '*loc_x*', say. It appears in the residual program as '`int loc_x`'. For the sake of readability, assume that scalar (dynamic) variables are bound to their own name.

Constraint 3.4 Dynamic variables shall be bound to their runtime address during specialization.

Consider specialization of the expressions '`ip = &y; *ip = 2`' (where '`ip`' is a static pointer to a dynamic object). In the generating extension, this code appears as

```
ip = &y; cmixAssign(*ip, "=", cmixInt(2))
```

where '`cmixInt()`' lift the static constant in the dynamic context. Operationally speaking, first the static pointer '`ip`' is assigned the address of the dynamic '`y`'. Next '`ip`' is dereferenced giving '`y = 2`', as desired.

Example 3.20 Consider the following contrived inline version of swap, implemented via pointers.

```
int x, y, *px = &x, *py = &y;
int temp;
/* swap */
temp = *px; *px = *py; *py = temp;
```

Suppose that '`x`' and '`y`' are dynamic. Specialization produces the following residual program

```
int x, y;
int temp;
temp = x; x = y; y = temp;
```

where no pointer manipulations take place. **End of Example**

Additional conditions have to be imposed on the usage of the address operator at specialization time when applied to arrays; this is described below.

3.6.2 Pointers and function arguments

Consider specialization of the '`swap()`' function.[28]

```
/* swap: swap the content of px and py */
void (int *px, int *py)
{
   int temp;
   temp = *px, *px = *py, *py = temp;
}
```

[28]See Kernighan and Ritchie [Kernighan and Ritchie 1988, page 95].

Suppose that '`int a,b`' are dynamic variables and consider specialization due to the call '`swap(&a,&b)`'. If the formal parameters are classified partially static, *i.e.* static pointers to dynamic objects, the net effect of the dereferencing (in the body) will be that the runtime addresses of '`a`' and '`b`' illegally are propagated into the (residual) body of '`swap()`'.

```
/* Illegal "residual" program for sway */
void swap_px_py()
{
   int temp;
   temp = a; a = b; b = temp;
}
```

To prevent this, specialization with respect to partially-static pointers is disallowed.

Constraint 3.5 Formal parameters of pointer type shall be completely static or completely dynamic.

It suffices to prohibit specialization with respect to pointers that point to non-local objects. The pointer analysis developed in Chapter 4 can implement this improvement. Furthermore, if the indirection of a partially-static pointer is dead in the body, the pointer can retain its status as static, see Section 3.10.

Example 3.21 Consider the following function that increments a pointer.

```
/* inc_ptr: increment integer pointer by one */
int *inc_ptr(int *p)
{
   return p + 1;
}
```

If '`p`' is a partially-static pointer, the above requirement forces '`p`' to be fully suspended. Since '`p`' not is dereferenced in the body of '`inc_ptr()`', it is permissible to classify the function static. The point is that '`*p`' is not used in the body, and that functions need not be specialized to dead values, see Section 3.12. **End of Example**

3.6.3 Pointers and arrays

Suppose that '`int a[N+1]`' is a statically indexed array of dynamic values.[29] The array can then be *split* into separate variables '`int a_0`',...,'a_N' during specialization, the objective being elimination of index calculations and a level of indirection.

Example 3.22 Suppose that a "polish calculator" uses a stack '`int s[STACK]`' to carry out the calculations. For example, the addition $x + y$ is computed by the instructions '`push(x); push(y); push(pop() + pop())`'. If the stack is statically indexed, specialization with respect to the "program" produces the residual program: '`s_0 = x, s_1 = y, s_0 = s_0 + s_1`', where '`s_0`' is the symbolic address of '`s[0]`' *etc.* **End of Example**

[29]Statically indexed means that all index expressions e in a[e] are static.

Naturally, if the index into an array is dynamic, the array cannot be split. More precisely, to split an array the following requirements must be fulfilled.

Constraint 3.6 An array '`a`' can be split provided:

- all index expressions into '`a`' are static, and
- no dynamic pointer may refer to '`a`', and
- the array '`a`' is not passed as actual argument to a function.

The conditions are justified as follows. An index expression '`a[`*e*`]`' corresponds to '`*(a + (`*e*`))`', and clearly the addition cannot be evaluated when e is dynamic. If a dynamic pointer may reference an array, the array cannot be split since *i*) the entry to which the pointer points may not be computable at specialization time and *ii*) the program may rely on the array being allocated as a consecutive block of memory, *e.g.* use pointer arithmetic. Finally, splitting an array passed as parameter to a function (or passing pointers to the individual variables in the residual program) introduces a severe call overhead.

Example 3.23 Consider an application of the address operator '`&a[3]`' where '`a`' is an array of dynamic elements. Even though '`a[3]`' evaluates to a dynamic value, the application can be classified static as the rewriting '`&a[3]`' $\equiv$ '`a + 3`' shows. On the other hand, in the case of a dynamic index e, '`&a[`*e*`]`' cannot be evaluated statically. **End of Example**

Binding time classification and specialization of runtime allocated arrays are similar to statically allocated arrays.

3.6.4 Address arithmetic

The C language allows a pointer to be assigned to another pointer, adding and subtracting a pointer and an integer, subtracting or comparing two pointers to members of the same array, and assigning and comparing to zero.[30] The hard part is the binding time separation. This is considered in Chapters 4 (pointer analysis) and 5 (binding-time analysis).

Example 3.24 The '`strcpy()`' function makes use of pointer arithmetic.

```
/* strcpy: copy t to s (K&R page 106) */
void strcpy(char *s, char *t)
{
   while (*s++ = *t++) ;
}
```

Suppose that '`s`' is dynamic and '`t`' is static. The two increment operators then have to be classified dynamic and static, respectively. The specialized program with respect to '`t = "ab"`' is shown below.

[30]Recall that pointer arithmetic only is allowed to shuffle a pointer around inside the same array, and one past the last element.

```
/* strcpy_t: Copy "ab" to s */
void strcpy_t(char *s)
{
   *s++ = 'a';
   *s++ = 'b';
   *s++ = '\0';
}
```

Assessment: the usability of this is doubtful. In practice a "specialized" function of the form

```
void strcpy_t(char *s)
{
   char *b = "ab";
   while (*a++ = *b++) ;
}
```

is preferable due to the reduced storage usage. **End of Example**

Binding time classification of the pointer versions of plus and minus differs from normal in that they can be applied to partially-static pointers. For example, if '`p`' is a static pointer to a dynamic object, '`p + 2`' can be classified static, since only the static part of '`p`' is needed to evaluate the addition. See Chapter 5 for details.

3.6.5 Character pointers and functions

In Section 3.3.1 we defined lift functions for constants of base type and strings. An example why it is useful to lift string constants is given below.

Example 3.25 The string function '`strcmp(char *, char *)`' is an external function, and must therefore be suspended if an argument is dynamic. The call '`strcmp("ab",b)`', where '`b`' is dynamic, is transformed to '`cmixECall("strcmp", cmixString("ab"), b)`', where '`cmixECall()`' returns a residual function call to '`strcmp()`'. **End of Example**

Lift of arbitrary character arrays is undesirable since it may duplicate data. Imagine for example an editor representing the edited file as a string! Section 3.10 considers externally allocated strings and string-functions.

3.6.6 Pointer arrays, pointers to pointers, multi-dimensional arrays

So far we have only employed one-level pointers. The techniques carry over to multi-level pointers without modification. For instance, a pointer '`int **p`' might be classified to be a static pointer to a static pointer to a dynamic object, and can hence be dereferenced twice during specialization. Naturally, it makes no sense to classify a pointer as "dynamic pointer to static pointer to dynamic objects", since the objects to which a pointer points must be present at runtime.

3.6.7 Pointers to functions

C has flexible rules for assignment of function addresses to pointers. We assume the standardized notation '`fp = &f`' for assignment and '`(*fp)()`' for call, whenever '`f`' is a function and '`fp`' a pointer of a suitable type, see Chapter 2.

Specialization with respect to function pointers is similar to, but simpler, than specialization with respect to higher-order values in functional languages. It is simpler in that C does not allow closures and partial function applications.

Consider a general application '`(*fp)(`e_1`,...,`e_n`)`', and differentiate between the following three situations:

1. The pointer '`fp`' is static, and hence points to a particular (generating-)function during specialization.

2. The pointer '`fp`' is dynamic, but is known to point to one of '$\{f_1, \ldots, f_n\}$'.

3. The pointer '`fp`' is dynamic, and only imperfect information about which functions it points is available.

Consider each case in turn.

If the function pointer is static, it can be dereferenced and the corresponding generating function called.[31] The call is transformed into

$\texttt{cmixICall((*fp)(}\overline{e_1}\texttt{,...,}\overline{e_m}\texttt{),}\overline{e_{m+1}}\texttt{,...,}\overline{e_n}\texttt{)}$

where e_{m+1},..., e_n are the dynamic arguments. Suppose that '`fp`' points to '`f`'. The residual call will then be '`f_res(`e'_{m+1}`,...,`e'_n`)`'.

A function pointer can (at most) point to all functions defined in the program.[32] Typically, however, it will only point to a small set of those. By the means of a *pointer analysis* the set of functions can be approximated. Assume that '`fp`' is a dynamic pointer and that it has been determined that it only can point to the functions '`f1()`',..., '`fn()`'. The idea is to specialize *all* the functions with respect to the static arguments, and defer the decision of which (specialized) function to call to run-time.

Example 3.26 Let '`fp`' be a function pointer, and '`f1()`' and '`f2()`' defined functions.

```
/* Example of dynamic function pointer */
fp =  dyn-exp? &f1 : &f2;
x = (*fp)();
```

During the specialization, the addresses of the defined functions are replaced by unique numbers assigned to '`fp`'. At the call site, a `switch` over the possible functions is generated.

[31] We ignore the case where the function is static.

[32] In this paragraph we consider a whole program; not a translation unit.

```
/* Specialized example of dynamic function pointer */
fp =  dyn-exp? 1 : 2;
switch (fp) {
 case 1: x = f1_spec(); /* Specialized version of f1() */
 case 2: x = f2_spec(); /* Specialized version of f2() */
}
```

Observe that the functions '`f1()`' and '`f2()`' usually will not appear in the residual program, so '`fp = &f1`' has no meaning. One function pointer assignment may cause several different specialized function to be called. **End of Example**

Since the actual function executed at runtime is unknown at specialization time, the potential functions may not perform any static side-effects.

Constraint 3.7 Functions (potentially) called via a function pointer shall not commit any static side-effects.

Finally we consider the case where the set of functions a function pointer may point to cannot be approximated. This can for example happen if the pointer is assigned entries of an externally defined array of functions.[33] The problem is that the function pointer may point to both an "unknown" function and a user-defined function.

Example 3.27 Suppose that '`int *cmp[]()`' is an array of function pointers and that '`int f()`' is a defined function.

```
/* Example code with a function pointer */
fp =  dyn-exp? cmp[ exp] : &f;
x = (*fp)();
```

Using a type cast, the unique "label" for '`f`' can be assigned '`fp`'.

```
/* Specialized example code */
fp =  dyn-exp? cmp[ exp] : (int (*)())1;
switch ((int)fp) {
   case 1:  x = f_spec(); /* Specialized version of f */
   default: x = (*fp)();  /* Call unknown function */
}
```

The `switch` tests whether the pointer points to a user defined function, otherwise it calls the "unknown" function. **End of Example**

Depending on the compiler, the `switch` may be converted to a jump-table during the final compilation.

Specialization with respect to dynamic function pointers can be summarized as follows. During execution of the generating extension:

1. expressions of the form '`&f`', where '`f`' is a defined function, are replaced by a unique constant (*e.g.* the address of the function), where, however,

[33]We now consider a program to consist of several translation units, where some are subject for specialization.

2. references to unknown functions are left unchanged.
3. Calls '`(*fp)()`' are specialized into a `switch` over the possible values of '`fp`', see Example 3.27.

The generating-extension transformation of an indirect call should be clear. Note that even though the residual code is not *strictly conforming* to the Standard, it is *conforming* [ISO 1990]. It is easy to see that it is actually portable.

The solution presented above is not optimal. It replaces an efficient construct (indirect call) by a potentially more expensive construct (switch statement). Our thesis is that the `switch` statements can be optimized by the compiler, and the gain obtain by specialization outweigh the extra call overhead. In the following we shall not consider specialization of function pointers with respect to both known and unknown functions.

3.7 Structures

A structure consists of one or more logically related variables which can be passed around as an entity. Classification of structs into either completely static or completely dynamic is likely to produce conservative results.

This section develops a more fine-grained binding-time assignment that allows automatic splitting of partially-static structs. Further, we investigate runtime memory allocation, and show how dynamic memory allocation can be replaced by static allocation. Finally we discuss unions and bit-fields.

3.7.1 Basics of structures

A structure is a collection of variables called *members*. For example,

```
struct point { int x, y, z; } pt;
```

defines a struct (and a variable '`pt`') useful for representation of coordinates.

A struct[34] where some but not necessarily all members are dynamic, is called a *partially-static struct*. We differentiate between the cases where the members of a struct can be accessed at specialization time, and where this is not the case. The former are called a *static* structs, the latter are said to be *dynamic*.

Constraint 3.8 The members of a dynamic struct shall all be dynamic.

A struct may for instance be classified dynamic if it is part of the unknown input to the subject program,[35] or it is returned by a function (see the next section). Notice that we attribute the classification to struct definitions — not to variables.[36] A struct index operator '.' applied to a static struct can be evaluated at specialization time. A dynamic

[34]In the following we are often sloppy and write a 'struct' for a 'variable of structure type', as is common practice.

[35]Normally, it is undesirable to change the type of the dynamic input.

[36]Recall that type separation is performed on representations of programs, cf. Section 2.3.

indexing $e.i$ is transformed into '`cmixStruct(`$\overline{e}$`, i)`', where $\overline{e}$ is the transformation of expression e.

The net effect is that static structs are *split* into separate variables, whereas the indexing into dynamic structs is deferred to runtime. The objective of struct splitting is elimination of a level of indirection and offset calculation.

Example 3.28 Suppose that '`struct point`' is a static struct where '`x`' is dynamic, and '`y`' and '`z`' are dynamic. Consider the following code:

```
struct point pt; /* static struct SxDxD */
pt.x = 2;        /* assignment to static member */
pt.y = 3;        /* assignment to dynamic member */
```

Specialization yields the following lines of code.

```
int pt_y, pt_z; /* the dynamic members of pt */
pt_y = 3;       /* assignment to dynamic member */
```

The struct variable '`pt`' has been split into the residual variable '`pt_y`' and '`pt_z`', both of type integer. **End of Example**

Dynamic struct members shall be bound to a runtime symbolic address as other dynamic variables.

Constraint 3.9 Static structs are split during the specialization such that dynamic members appear as separate variables in the residual program.

We consider the interaction between structs and pointers below.

Example 3.29 Another strategy is to "narrow" the definition of partially-static structs to the dynamic members. During specialization, static members are accessed, whereas code is generated for references to dynamic members. Structs are not split. When compilers pass structs via references, this approach may outperform struct splitting. We shall not, however, pursue this any further. **End of Example**

3.7.2 Structures and functions

A struct can be passed to a function in two ways: call-by-value or via a pointer. The ANSI C Standard allows functions to return structs.

Consider the following function that returns a struct.

```
/* make_point: create point given coordinates */
struct point make_point(int x, int y, int z)
{
   struct point pt;
   pt.x = x; pt.y = y; pt.z = z;
   return pt;
}
```

Suppose that 'x' is static, and 'y' and 'z' are dynamic. Since 'pt' is returned by the function, it cannot be split: it must be classified dynamic. To see why, consider a call 'pt = make_point(1,2,3)', and suppose that the member 'x' is static. The variable 'pt.x' would then need updating after specialization of 'make_point()', which is undesirable.

Constraint 3.10 Structs returned by a function shall be dynamic.

Consider now the opposite: structs passed call-by-value to functions. The 'length()' function below receives a 'struct point'.

```
/* length: compute length from origin to point */
double length(struct point pt)
{
   return sqrt(pt.x * pt.x + pt.y * pt.y + pt.z * pt.z);
}
```

To have a clear separation of static and dynamic arguments, we prohibit passing of partially-static structs.

Constraint 3.11 A struct type used in a parameter definition shall have either completely static members or completely dynamic members. In the latter case the struct shall be classified dynamic.

Information lost due to the above requirement can be revealed by the passing of struct members as separate variables. Naturally, care must be taken not to change the passing semantics, *e.g.* by passing directly an array previously encapsulated in a struct.

Example 3.30 To avoid 'struct point' to be classified as dynamic (since it is passed to the function 'length()'), we apply the following transformation, which clearly can be automated.

```
/* Transformed version of length(struct point pt) */
double length(int x, int y, int z)
{
   struct point pt = { x, y, z };
   return sqrt(pt.x * pt.x + pt.y * pt.y + pt.z * pt.z);

}
```

The use of the local variable 'pt' is needed if the address of the (original) parameter is taken, see below. **End of Example**

Precisely, by applying the following transformation to subject programs, partially-static structures can be "passed" to functions. The transformation can only be applied when none of the members are of array type.

- A call 'f(s)' is changed to 'f(s.x,s.y...,s.z)', where 'x','y',...,'z' are the members of 's'. The function definition must changed accordingly.

- In the body of the function, a local variable of the original struct type is introduced and initialized via parameters.[37]

This transformation can obviously be automated.

3.7.3 Pointers to structures

This section considers pointers to structs created via the address operator &. Runtime memory allocation is the subject of the section below.

When the address operator is applied to a static struct, the result is a pointer that can be used to access the members. When the address operator is applied to a dynamic struct, it can only be used to reference the struct object.

Example 3.31 Assume the usual binding-time assignment to '`struct point`'.

```
struct point pt1, pt2, *ppt;
pt1.x = 1; pt1.y = 2; pt1.z = 3;   /* 1 */
ppt = &pt1;                        /* 2 */
pt2.x = ppt->x; pt2.y = ppt->y;    /* 3 */
pt2 = *ppt;                        /* 4 */
```

Specialization proceeds as follows. In line 1, the assignment to '`x`' is evaluated, and code is generated for the rest. In line 2, a static pointer to '`pt1`' is created, and is dereferenced in line 3. Consider line 4. The assignment is between partially-static structs. Applying the transformation described in Section 3.3 splits the assignments into a static and a dynamic part. Thus, the following residual code will be generated.

```
int pt1_y, pt1_z, pt2_y, pt2_z;
pt1_y = 2; pt1_z = 3;            /* 1 */
pt2_y = pt1_y;                   /* 3 */
pt2_y = pt1_y; pt2_z = pt1_z;    /* 4 */
```

If the assignment in line 4 is not split, '`struct point`' inevitably must be reclassified dynamic. **End of Example**

Notice that structs cannot be lifted, so a struct appearing in a dynamic contexts must be classified dynamic.

Constraint 3.12 Structs referred to by a dynamic pointer shall be classified dynamic.

Consider now the passing of a struct pointer to a function. For example, suppose that the function '`length`' is changed into '`length(struct point *ppt)`' that accepts a pointer to a struct. As in the case of pointers to other objects, this can cause (dynamic) objects lexically out of scope to be propagated into the body of the called function. To avoid this, pointers to partially-static structs are not allowed.

[37]This transformation increases the number of parameters to a function, and may thus represent a call overhead. In the case of structs with a few members which can be are passed in registers, the transformation has no effect, though.

Constraint 3.13 A parameter of type "pointer to struct" shall be dynamic, if the struct contains a dynamic member.

Since a struct referred to by a dynamic pointer itself must be classified dynamic; this fully suspends the struct.

Example 3.32 Suppose that the function '`length(struct point pt)`' takes the address of the parameter '`pt`', and uses it to compute the result. Suppose that the parameter splitting transformation as outlined in the previous section is applied.

```
int length(int x, int y, int z)
{
   struct point pt = { x, y, z };
   struct point *ppt;
   ppt = &pt;
   return sqrt(ppt->x * ppt->x + ppt->y * ppt->y + ppt->z * ppt->z);
}
```

This function specializes into the same residual program as shown before (modulo copy propagation elimination of '`pt_y = y`' and '`pt_z = z`'). Notice that introduction of '`pt`' is necessary for the address operator to be applicable. **End of Example**

3.7.4 Self-referential structures

A struct definition can refer to itself by the means of a pointer. Recall from Chapter 2 that recursively defined structures are 1-limited by type definition. We convey this restriction to binding-time assignments. For example,

```
struct nlist { struct nlist *next; char *name, *defn; }
```

defines a struct representing nodes in a list. If the binding time of '`struct nlist`' is BT, the binding time of '`next`' must be "static pointer to BT", or "dynamic pointer".

Constraint 3.14 Binding time assignments to self-referential structs are 1-limited.

The actual choice of k-limit has no influence on the rest of this chapter.

3.7.5 Runtime allocation of structures

In C, all (dynamic) runtime memory allocation is performed by the means of the library function '`malloc()`', or one of its variants.[38] For example, the call

```
(struct nlist *)malloc(sizeof(struct nlist))
```

returns a pointer to storage to represent an object of type '`struct nlist`'. However, nothing prevents a programmer from allocating twice as much memory as really needed, or to convert the returned pointer to a '`struct point *`' and use it as such. Clearly, this allocation scheme is too liberal for automatic program analysis and transformation.

We shall therefore suspend all calls to '`malloc`' (and its cousins), and introduce a special allocation function '`alloc()`' with a more restricted semantics.

[38]In the case of Unix, these functions all use the 'sbrk' operating system call to get a chunk of storage.

Constraint 3.15 The call '`alloc(TypeName)`' shall perform the following:

- Return a pointer (suitably converted) to a sufficient chunk of memory to represent an object of the indicated type.
- Inform the memory manager in the generating extension that the storage has been allocated.
- In the case where the type is a partially-static struct, add the declarations of its dynamic members as global variables to the residual program.

An '`alloc()`' will be performed at specialization time, *i.e.* memory will be allocated when the generating extension is run, if it appears in a static context, *e.g.* if the result is assigned a static pointer.

We also assume a function '`delete()`' with same functionality as '`free()`'. The handling of runtime memory in the generating extension is discussed in detail in Section 3.10.

Example 3.33 Consider the following program fragment which allocates a node and initializes it. Assume that '`next`' is a static pointer, that '`name`' is static, and that '`defn`' is dynamic.

```
struct nlist *np;
np = alloc(struct nlist);
np->next = NULL;
np->name = strdup("fubar");
np->defn = strdup("foobar");
printf("Name is %s, definition is %s", np->name,np->defn);
```

The call to '`alloc()`' (static) causes some memory to be allocated, and the definition of '`alloc1_defn`' to be added to the residual program. The assignment '`np->next = NULL`' initializes the static '`next`' with `NULL`, and '`np->name = strdup("fubar")`' the '`name`' field with '`"fubar"`'. The final assignment is dynamic. This yields the following residual program.

```
char *alloc1_defn;
alloc1_defn = strdup("foobar");
printf("Name is %s, definition is %s", "foobar", alloc1_defn);
```

Notice that the static pointers '`next`' are eliminated, and all the pointer traversal is performed at specialization time. **End of Example**

Automatic detection of '`malloc()`' calls that safely can be replaced by '`alloc()`' requires an analysis of the heap usage. We assume it has been done prior to the generating extension transformation. An '`alloc()`' variant must exist for each base type, pointer type, and user-defined type. These can be generated automatically of the basis on type definitions.

```
struct nlist {
   struct nlist *next;  /* next entry in the chain */
   char *name;          /* defined name */
   char *defn;          /* replacement text */
} *hashtab[HASHSIZE];
/* lookup: look for s in hashtab */
struct *nlist lookup(char *s)
{
   struct nlist *np;
   for (np = hashtab[hash(s)]; np != NULL; np = np->next)
      if (strcmp(s, np->name) == 0) return np; /* found */
   return NULL; /* not found */
}
/* install: put (name,defn) into hashtab */
int install(char *name, char *defn)
{
   struct nlist np; unsigned hashval;
   if ((np = lookup(name)) == NULL) { /* not found */
      np = alloc(struct nlist);
      if (np == NULL || (np->name = strdup(name)) == NULL) return 0;
      hashval = hash(name);
      np->next = hashtab[hashval];
      hashtab[hashval] = np;
   } else /* already there */
      delete (np->defn);
   if ((np->defn = strdup(defn)) == NULL) return 0;
   return 1;
}
```

Figure 22: Functions for text substitution

3.7.6 Replacing dynamic allocation by static allocation

We demonstrate specialization of a set of functions implementing text substitution. The example stems from Kernighan and Ritchie [Kernighan and Ritchie 1988, page 143], and is reproduced in Figure 22. The main result of this example is the replacement of (dynamic) runtime allocation by static allocation.

A few changes to the original program have been made. The call to '`malloc()`' has been replaced by a call to '`alloc()`', and the '`install()`' function changed to return an integer instead of the pointer to the installed node.[39]

We consider specialization of the following sequence of statements:

```
struct nlist *np;
install("fubar", "foobar");
if ((np = lookup("fubar")) != NULL)
   printf("Replace %s by %s", "fubar", np->defn);
```

where it is assumed that the second argument to '`install()`' is unknown.

[39]Both changes are essential. The former to assure memory is allocated during the specialization; the latter to prevent 'struct nlist' from becoming dynamic.

```
char *alloc1_defn;
int install_fubar(char *defn)
{
   if ((alloc1_defn = strdup(defn)) == NULL) return 0;
   return 1;
}

install_fubar("foobar");
printf("Replace %s by %s", "fubar", alloc1_defn);
```

Figure 23: Specialized version of Figure 22

By examination of the program text we see that '`struct nlist`' is a (partially) static struct with '`next`' and '`name`' static and '`defn`' dynamic. Even though the function '`lookup()`' manipulates pointers to partially-static structs, it can be classified static.[40] In the definition of '`install()`', only the references to '`defn`' are dynamic.

The specialized program due to the call '`install("fubar","foobar")`' is given in Figure 23. During specialization, the calls to '`lookup()`' have been evaluated statically. Further, the runtime allocation has been replaced by static allocation in the specialized program. The test in the `if` statement has been evaluated to "true", since '`lookup()`' is static, and the pointer reference '`np->defn`' reduced to '`alloc1_defn`'.

3.7.7 Unions

A union resembles a struct but with a major operational difference: at a time, a union can only represent the value of one member, and henceforth only memory to represent the largest member needs to be allocated.

Suppose that a union was split into separate variables, *e.g.*

```
union  U { int x, y, z; }
```

This would increase the storage usage in the residual program considerably — naturally depending on the number of static members in the union. To avoid this we suspend all unions.

Constraint 3.16 Unions shall be classified dynamic.

Example 3.34 Splitting of union is not straightforward. The Standard specifies that if a union contains members of struct type that have an initial member sequence in common, *e.g.* '`tag`':

```
union Syntax {
   struct { int tag; ... } Expr_stmt;
   struct { int tag; ... } If_stmt;
}
```

[40]Technically, the member 'defn' of 'struct nlist' is not used in the body of lookup.

when it is legal to access a shared member (‘`tag`’) through another struct member (*e.g.* `If_stmt`) than the one that defined the field (*e.g.* `Expr_stmt`).

This implies that *e.g.* the field ‘`tag`’ cannot be split into separate variables, and so must be a “shared” variable. **End of Example**

In our experience this restriction is not too strict in practice. More experiments are needed to clarify this, however.

3.7.8 Bit-fields

Bit-fields are used to restrict the storage of a member, but have otherwise no influence on program specialization. Thus, bit-field specifications are can be translated unchanged to both the generating-extension and — from there — to the residual programs. Notice that since generating extensions are executed directly, no representation problems are encountered, *e.g.* due to programs relying on a particular layout of a struct.

3.8 Input and output

In the previous sections it was assumed that static input was delivered via parameters to the generating extension’s goal function. This section is concerned with I/O: reading and writing standard input/output, and files. Moreover, we discuss the standard-library functions, *e.g.* ‘`pow()`’, which are declared ‘`extern`’ in the standard header files.

3.8.1 Standard input and output

Many (small) C programs read data from standard input and write the result to the standard output. Naturally, no output shall occur when a generating extension is executed.

Constraint 3.17 All writes to standard output shall be suspended.

When reading of input streams are allowed during specialization, care must be taken to avoid duplication of I/O-operations. The potential risk is illustrated in the code below.

```
int d; float f;
if ( dyn-exp )
   scanf("%d", &d);
else
   scanf("%f", &f);
```

Specialization of this code will cause the ‘`scanf()`’ to be called twice contrary to normal execution.

User annotations can indicate which input calls that desirably should be performed during the execution of the generating extension. The control-dependence analysis developed in Chapter 6 can be employed to detect and warn about dubious read calls. We leave this extension to future work.

Constraint 3.18 All reads from standard input shall be suspended.

File access and reading of static input from files are discussed below.

```
/* miniprintf: mini version of printf */
void minprintf(char *fmt, void *values[])
{
   int arg = 0;
   char *p, *sval;
   for (p = fmt; *p; p++) {
      if (*p != '%') {
         putchar(*p);
         continue;
      }
      switch (*++p) {
      case 'd':
         printf("%d", *(int *)values[arg++]);
         break;
      case 'f':
         printf("%f", *(double *)values[arg++]);
         break;
      case 's':
         for (sval = *(char **)values[arg++]; *sval; sval++)
            putchar(*sval);
         break;
      default:
         putchar(*p);
         break;
      }
   }
}
```

Figure 24: A mini-version of `printf()`

3.8.2 Case study: formatted output — printf

The output function '`printf()`' (and its friends) takes a format string and a number of values, and converts the values to printable characters. Thus, '`printf()`' is a small interpreter: the format string is the "program", and the values are the "input". By specializing with respect to the format string, the interpretation overhead can be removed.[41] [Consel and Danvy 1993].

Figure 24 contains a mini-version of the '`printf()`' library function; the program stems from Kernighan and Ritchie [Kernighan and Ritchie 1988, Page 156]. A minor change has been made: instead of using a variable-length argument list, it takes an array of pointers to the values to be printed. The reason for this change will become apparent in the next section.

Suppose that '`fmt`' is static and '`values`' is dynamic. The `for` loop is controlled by the format string, and can be unrolled. The tests in both the `if` statements and the `switch` statements are also static. The output calls are all dynamic, cf. the discussion in Section 3.8.1.

Let the static '`fmt`' string be '`"%s = %d\n"`'. The specialized version is shown in

[41]This example was suggested to us by Olivier Danvy.

```
void minprintf_fmt(void *values[])
{
   char *sval;
   for (sval = *(char **)values[0]; *sval; sval++)
      putchar(*sval);
   putchar(' ');
   putchar('=');
   putchar(' ');
   printf("%d", *(int *)values[1]);
   putchar("\n");
}
```

Figure 25: The printf() function specialized with respect to `"%s = %d\n"`

Figure 25. Running the original and the specialized program 100,000 times on the dynamic input `"value"` and 87 takes 2.12 and 1.70 seconds, respectively. That is, the speedup is 1.2. Naturally, the speedup depends on the static input; larger static input yields larger speedup.

3.8.3 Variable-length argument lists

By the means of the ellipsis construct '`...`', functions can be defined to taken a variable-length argument list. For example, the declaration '`void printf(char *fmt, ...)`' specifies that '`printf()`' takes an arbitrary number of arguments besides a format string.

When all calls to a function with a variable-length argument-list are known, *specialized* versions of the function can be created by a simple preprocess, and the program can be specialized normally. Suppose that the function is part of a library, and hence call-sites are unknown.

This implies that the type and number of arguments first will be known when the generating extension is run. This complicates the memory management[42] and renders binding-time analysis hard.

Moreover, according to the standard definition of '...', there must be at least one argument in front of an ellipsis.

Constraint 3.19 Ellipses and the parameter before shall be suspended to runtime.

In the case of the '`printf()`' function studied in the previous section, this would have caused '`fmt`' to be suspended, rendering specialization trivial.

We believe that variable-length argument lists are rare in C programs, so suspension seems reasonable. We have not implemented this feature into *C-Mix*, and will ignore it in the rest of this thesis.

3.8.4 File access

So far we have courageously assumed that static values are delivered through goal parameters. In practice, inputs are read from files, generated by functions *etc.* We discuss

[42]The generating of copying functions depend on the availability of type information.

reading of files, and more broadly how static input is delivered at specialization time. The material in this section is rather pragmatically oriented.

Reading of files is similar to scanning of standard input, Section 3.8.1. User annotation, for instance in the form of a 'cmix-fscanf', can be employed to indicate streams to be read during specialization. To avoid duplication of I/O-operations, warnings about static reads under dynamic control must be given to the user. We do not consider this an essential feature, for reasons to be discussed below, and leave it to future work.

A program normally consists of several translation units, or modules, each implementing different aspects of the overall system. Typically, there is a module for reading and initialization of data structures, and a module implementing the "computation". Normally, specialization is applied to the "computation module" — nothing or only little is gained by specialization of input routines.[43] A program system is specialized as follows. The computation module is binding time analyzed and transformed into a generating extension. The resulting generating extension is linked together with the I/O-functions that open, read and close files. Thus, there is no real need for reading of files during specialization.

3.8.5 Error handling

Partial evaluation is more "strict" than normal order evaluation.[44] The reason is that a partial evaluator executes both the branches of a dynamic `if` while standard execution executes only one.

Example 3.35 Consider the following program fragment where a '`lookup()`' function returns whether a record has been found, and in the affirmative case sets the pointer '`p`' to point to it.

```
found = lookup(name, &p);
if (found) printf("Number %d\n", p->count);
else       printf("Not found\n");
```

Partial evaluation of the fragment, where '`found`' is dynamic and '`p`' is static, may go wrong. **End of Example**

Errors at specialization time can be detected by guarding the potential expressions. For example, *safe* pointers [Edelson 1992] can detect dereferencing of NULL-pointers and trap a handler. We have not implemented this in the current version of *C-Mix*, and we suspect it will be expensive.

3.8.6 Miscellaneous functions

The standard library is an important part of C. For example, the library '`libm`' defines the functions declared in '`<math.h>`'. All library functions are declared '`extern`'.

[43]A remarkable exception is that 'scanf' can be specialized similarly to 'printf', eliminating the interpretation overhead.

[44]Partial evaluation has been termed "hyper strict" in the literature.

$$\frac{\vdash^{tdef} T_i \Rightarrow T'_i \quad \vdash^{decl} d_i \Rightarrow d'_i \quad \vdash^{fun} f_i \Rightarrow f'_i}{\begin{array}{ll}\vdash^{gegen} & \langle T_1;\ldots;T_m, d_1;\ldots;d_n, f_1\ldots f_l\rangle \Rightarrow \\ & T'_1;\ldots;T'_m \\ & d'_1;\ldots;d'_n \\ & f'_1\cdots f'_l \\ & \texttt{int generate(}d_1^{goal},\ldots,d_k^{goal}\texttt{)} \\ & \{ \\ & \quad \text{Declare-residual-struct;} \\ & \quad \text{Declare-residual-globals;} \\ & \quad f^{goal}\texttt{(}d_1^{goal},\ldots,d_n^{goal}\texttt{);} \\ & \quad \texttt{unparse();} \\ & \quad \texttt{return 0;} \\ & \}\end{array}}$$

Figure 26: Generating-extension transformation

Recall from Section 3.5 that calls to externally defined functions are suspended. This has the unfortunate effect that a static call '`pow(2,3)`' is suspended, and not replaced by the constant 8, as expected.

Example 3.36 The reason for suspending external functions is that they may side-effect global variables. This is indeed the case in the '`pow()`' function.

```
#include <math.h>
int main(void)
{
   double p = pow(2.0, 3.0);
   if (errno) fprintf(stderr, "Error in pow");
   else printf("Result %f", p);
}
```

Many library functions report errors via the (external) variable '`errno`'. **End of Example**

Static calls to external functions that do not side-effect global variables can be evaluated at specialization time (provided their definitions are present).

Definition 3.2 *A function that does not side-effect non-local variables is called* pure. □

Pure functions can be detected in two ways: either by user annotations, or automatically. In Section 6 we outline an analysis to determine pure functions. In examples we use a specifier '`pure`' to indicate pure functions.

Example 3.37 As mentioned above, many functions declared in '`<math.h>`' are not pure since they return errors in the global variable '`errno`'. However, by redefining the functions to invoke the '`matherr()`' function, the problem can be circumvented.

Thus, by declaring the power function '`extern pure double pow(double, double)`', the call '`pow(2.0, 2.0)`' will be evaluated at specialization time. **End of Example**

$$\frac{\vdash^{tdef} T \Rightarrow T' \quad \vdash^{decl} d_i \Rightarrow d'_i \quad \vdash^{stmt} S_j \Rightarrow S'_j}{\begin{array}{l}\vdash^{fun} \ T\ f(d_1,\ldots,d_m)\{\ d_{m+1},\ldots,d_n\ S_1,\ldots,S_k\} \ \Rightarrow \\ \qquad T'\ f(d'_1,\ldots,d'_m)\{\ d'_{m+1},\ldots,d'_n\ S'_1,\ldots,S'_k\}\end{array}}$$

$$\frac{\vdash^{decl} d_i \Rightarrow d'_i \quad \vdash^{stmt} S_i \Rightarrow S'_i}{\vdash^{fun} \ \underline{T\ f(d_1,\ldots,d_m)\{d_{m+1},\ldots,d_n\ S_1,\ldots,S_k\}} \Rightarrow}$$

```
Code $f(d'_1,\ldots,d'_m)$
{
    $d'_{m+1},\ldots,d'_n$
    cmixStateDesc locals[] =$\{x_1,\ldots,x_n\}$;/* $d'_i \equiv x_i\ T'_i$*/
    if (seenB4()) return cmixFun();
    cmixPushState(locals);
    cmixPushFun();
    Define-residual-variables;
    cmixPendinsert(&&entry);
  cmixPendLoop:
    if (cmixPP = cmixPending()) {
      lab = cmixRestore();
      cmixLabel(lab);
      goto *cmixPP;
    } else {
      cmixPopFun();
      cmixPopState();
      cmixFun();
    }
  entry:
    $S'_1,\ldots,S'_l$
}
```

Figure 27: Transformations of functions

3.9 Correctness matters

It is crucial that program transformations preserve the semantics of subject programs. It is fatal if an optimized program possesses a different observational behaviour than the original program. A main motivation for adopting the generating-extension approach was to assure correctness of residual programs.

In this section we summarizes the generating-extension transformation. We make no attempt at proving the correctness (Definition 3.1). However, we hope that it is clear that correctness of the generating-extension transformation is easier to establish than the correctness of a symbolic evaluator.

The Figures 26, 27, 28, 29, 30, 31, and 32, summarize the generating-extension transformation.

Input is an annotated program where dynamic constructs are indicated by an underline. For simplicity we assume that lifting of static values in dynamic contexts is denoted explicitly by the means of '`lift`' functions. In practice, application sites for lift functions would be discovered during the transformation. The rule for the address operator does not incorporate function identifiers. Further, we assume that dynamic indirect calls are lifted to the statement level. Finally, we have not shown "unparsing" of abstract declarations

$$
\begin{array}{lll}
\text{[decl]} & \dfrac{\vdash^{type} T \Rightarrow T'}{\vdash^{decl} T\ x \Rightarrow T'\ x} & \vdash^{decl} \underline{\texttt{extern}\ T\ x} \Rightarrow \texttt{extern}\ T\ x \\[2ex]
\text{[base]} & \vdash^{type} \langle \tau_b \rangle \Rightarrow \langle \tau_b \rangle & \vdash^{type} \underline{\langle \tau_b \rangle} \Rightarrow \langle \texttt{Code} \rangle \\[2ex]
\text{[struct]} & \vdash^{type} \langle \texttt{struct S} \rangle \Rightarrow \langle \texttt{struct S} \rangle & \vdash^{type} \underline{\langle \texttt{struct S} \rangle} \Rightarrow \langle \texttt{Code} \rangle \\[2ex]
\text{[pointer]} & \dfrac{\vdash^{type} T \Rightarrow T'}{\vdash^{type} \langle * \rangle\, T \Rightarrow \langle * \rangle\, T'} & \vdash^{type} \underline{\langle * \rangle} \Rightarrow \langle \texttt{Code} \rangle \\[2ex]
\text{[array]} & \dfrac{\vdash^{type} T \Rightarrow T'}{\vdash^{type} \langle [n] \rangle\, T \Rightarrow \langle [n] \rangle\, T'} & \vdash^{type} \underline{\langle [n] \rangle} T \Rightarrow \langle \texttt{Code} \rangle \\[2ex]
\text{[fun]} & \dfrac{\begin{array}{l}\vdash^{decl} d_i \Rightarrow d_i' \\ \vdash^{type} T \Rightarrow T'\end{array}}{\vdash^{type} \langle (d_i) \rangle\, T \Rightarrow \langle (d_i') \rangle\, T'} & \vdash^{type} \underline{\langle (d_i) \rangle} T \Rightarrow \langle \texttt{Code} \rangle
\end{array}
$$

Figure 28: Transformation of declarations

$$
\begin{array}{lll}
\text{[struct]} & \dfrac{\vdash^{decl} d_i \Rightarrow d_i'}{\vdash^{tdef} \texttt{struct}\ S\ \{d_i\} \Rightarrow \texttt{struct}\ S\ \{d_i'\}} & \vdash^{tdef} \underline{\texttt{struct}\ S}\ \{\ d_i'\} \Rightarrow; \\[2ex]
\text{[union]} & \dfrac{\vdash^{decl} d_i \Rightarrow d_i'}{\vdash^{tdef} \texttt{union}\ U\ \{d_i\} \Rightarrow \texttt{union}\ U\ \{d_i'\}} & \vdash^{tdef} \underline{\texttt{union}\ U}\ \{\ d_i'\} \Rightarrow; \\[2ex]
\text{[enum]} & \vdash^{tdef} \texttt{enum}\ E\{x = e\} \Rightarrow \texttt{enum}\ E\{x = e\} &
\end{array}
$$

Figure 29: Transformation of type definitions

to concrete syntax.

[call]

$$\frac{\vdash^{exp} e_i \Rightarrow e'_i}{\begin{array}{l}\vdash^{stmt} \underline{x\ \texttt{=}\ e_0\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}} \Rightarrow \\ \quad \{ \\ \quad\quad \texttt{switch((int)}e'_0\texttt{)\{} \\ \quad\quad\quad\quad \texttt{case 1:} \\ \quad\quad\quad\quad\quad \texttt{cmixAssign(}x\texttt{,"=",} \\ \quad\quad\quad\quad\quad\quad \texttt{cmixCall(}f_1\texttt{(}e'_s\texttt{),}e'_d\texttt{))} \\ \quad\quad\quad\quad \texttt{case n:} \\ \quad\quad\quad\quad\quad \texttt{cmixAssign(}x\texttt{,"=",} \\ \quad\quad\quad\quad\quad\quad \texttt{cmixCall(}f_n\texttt{(}e'_s\texttt{),}e'_d\texttt{))} \\ \quad\quad\quad\quad \texttt{default:} \\ \quad\quad\quad\quad\quad \texttt{cmixAssign(}x\texttt{,"=",} \\ \quad\quad\quad\quad\quad\quad \texttt{cmixPCall(}e'_0\texttt{(}e'_s\texttt{),}e'_d\texttt{))} \\ \quad\quad \} \\ \quad \}\end{array}}$$

where e_o calls $f_1,\ldots,f_n$

[empty] $\vdash^{stmt} ; \Rightarrow ;$

[exp]

$$\frac{\vdash^{exp} e \Rightarrow e'}{\vdash^{stmt} e; \Rightarrow e'} \qquad \frac{\vdash^{exp} e \Rightarrow e'}{\vdash^{stmt} \underline{e}; \Rightarrow \texttt{cmixExpr(}e'\texttt{);}}$$

[if]

$$\frac{\begin{array}{l}\vdash^{exp} e \Rightarrow e' \\ \vdash^{stmt} S_1 \Rightarrow S'_m \quad \vdash^{stmt} S_2 \Rightarrow S'_n\end{array}}{\begin{array}{l}\vdash^{stmt} \texttt{if (}e\texttt{)}\ S_m\ \texttt{else}\ S_n \Rightarrow \\ \quad\quad \texttt{if (}e'\texttt{)}\ S'_m\ \texttt{else}\ S'_n\end{array}}$$

$$\frac{\vdash^{exp} e \Rightarrow e' \quad \vdash^{stmt} S_m \Rightarrow S'_m \quad \vdash^{stmt} S_n \Rightarrow S'_n}{\begin{array}{l}\vdash^{stmt} \underline{\texttt{if (}e\texttt{)}\ S_m\ \texttt{else}\ S_n} \Rightarrow \\ \quad \{ \\ \quad\quad i\texttt{: cmix\_test = }e'\texttt{;} \\ \quad\quad \texttt{cmixIf(cmix\_test,} \\ \quad\quad\quad\quad \texttt{cmixPendinsert(\&\&}m\texttt{),} \\ \quad\quad\quad\quad \texttt{cmixPendinsert(\&\&}n\texttt{));} \\ \quad\quad \texttt{goto cmixPendLoop;} \\ \quad\quad m:\ S'_m; \\ \quad\quad \texttt{cmixGoto(cmixPendinsert(\&\&}l\texttt{));} \\ \quad\quad \texttt{goto cmixPendLoop;} \\ \quad\quad n:\ S'_n \\ \quad\quad \texttt{cmixGoto(cmixPendinsert(\&\&}i\texttt{));} \\ \quad\quad \texttt{goto cmixPendLoop;} \\ \quad\quad l:; \\ \quad \}\end{array}}$$

[switch]

$$\frac{\vdash^{exp} e \Rightarrow e' \quad \vdash^{stmt} S \Rightarrow S'}{\begin{array}{l}\vdash^{stmt} \texttt{switch (}e\texttt{)}\ S \Rightarrow \\ \quad\quad \texttt{switch (}e'\texttt{)}\ S'\end{array}} \qquad \frac{\vdash^{exp} e \Rightarrow e' \quad \vdash^{stmt} S \Rightarrow S'}{\vdash^{stmt} \underline{\texttt{switch (}e\texttt{)}\ S} \Rightarrow \text{Similar to } \texttt{if}}$$

[case]

$$\frac{\vdash^{tstmt} S \Rightarrow S'}{\begin{array}{l}\vdash^{stmt} \texttt{case}\ e\texttt{:}\quad S \Rightarrow \\ \quad\quad \texttt{case}\ e\texttt{:}\quad S'\end{array}} \qquad \frac{\vdash^{tstmt} S \Rightarrow S'}{\vdash^{stmt} \underline{\texttt{case}\ e\texttt{:}\quad S} \Rightarrow \text{Similar to } \texttt{if}}$$

[default]

$$\frac{\vdash^{stmt} S \Rightarrow S'}{\begin{array}{l}\vdash^{stmt} \texttt{default:}\quad S \Rightarrow \\ \quad\quad \texttt{default:}\quad S'\end{array}} \qquad \frac{\vdash^{stmt} S \Rightarrow S'}{\vdash^{stmt} \underline{\texttt{default:}\quad S} \Rightarrow \text{Similar to } \texttt{if}}$$

Figure 30: Transformation of statements (part 1)

[while]

$$\frac{\vdash^{exp} e \Rightarrow e' \quad \vdash^{smt} S_m \Rightarrow S'_m}{\vdash^{stmt} \begin{array}[t]{l}\texttt{while (}e\texttt{) } S_m \Rightarrow\\ \texttt{while (}e'\texttt{) } S'_m\end{array}}$$

$$\frac{\vdash^{exp} e \Rightarrow e' \quad \vdash^{stmt} S_m \Rightarrow S'_m}{\vdash^{stmt} \begin{array}[t]{l}\underline{\texttt{while (}e\texttt{) } S_m} \Rightarrow\\ \{\\ \quad l:\ \texttt{cmix\_test = }e'\texttt{;}\\ \quad \texttt{cmixIf(cmix\_test,}\\ \qquad\quad \texttt{cmixPendinsert(}m\texttt{),}\\ \qquad\quad \texttt{cmixPendinsert(}n\texttt{));}\\ \quad \texttt{goto cmixPendLoop;}\\ \quad m:\ S'_m\\ \quad \texttt{cmixGoto(cmixPendinsert(}l\texttt{));}\\ \quad \texttt{goto cmixPendLoop;}\\ \quad n:;\\ \}\end{array}}$$

[do]

$$\frac{\vdash^{exp} e \Rightarrow e' \quad \vdash^{stmt} S \Rightarrow S'}{\vdash^{stmt} \begin{array}[t]{l}\texttt{do } S \texttt{ while (}e\texttt{)} \Rightarrow\\ \texttt{do } S' \texttt{ while (}e'\texttt{)}\end{array}}$$

$$\frac{\begin{array}{l}\vdash^{exp} e \Rightarrow e'\\ \vdash^{stmt} S \Rightarrow S'\end{array}}{\vdash^{stmt} \begin{array}[t]{l}\underline{\texttt{do } S \texttt{ while (}e\texttt{)}} \Rightarrow\\ \{\\ \quad l:\ S'\\ \quad \texttt{cmix\_test = }e'\texttt{;}\\ \quad \texttt{cmixIf(cmix\_test,}\\ \qquad\quad \texttt{cmixPendinsert(}l\texttt{),}\\ \qquad\quad \texttt{cmixPendinsert(}m\texttt{))}\\ \quad \texttt{goto cmixPendLoop;}\\ \quad m:;\\ \}\end{array}}$$

[for]

$$\frac{\vdash^{stmt} e_i \Rightarrow e'_i \quad \vdash^{stmt} S \Rightarrow S'}{\vdash^{stmt} \begin{array}[t]{l}\texttt{for (}e_1\texttt{;}e_2\texttt{;}e_3\texttt{) } S \Rightarrow\\ \{\\ \quad e'_1;\\ \quad \texttt{while (}e'_2\texttt{) \{}\\ \qquad\quad S'\\ \qquad\quad e'_3\\ \qquad \}\\ \quad \}\end{array}}$$

$$\frac{\vdash^{exp} e_i \Rightarrow e'_i \quad \vdash^{stmt} S \Rightarrow S'}{\vdash^{stmt} \begin{array}[t]{l}\underline{\texttt{for (}e_1\texttt{;}e_2\texttt{;}e_3\texttt{) } S} \Rightarrow\\ \{\\ \quad \texttt{cmixExpr(}e'_1\texttt{);}\\ \quad l\texttt{: cmix\_test = }e'_2\\ \quad \texttt{cmixIf(cmix\_test,}\\ \qquad\quad \texttt{cmixPendinsert(}m\texttt{),}\\ \qquad\quad \texttt{cmixPendinsert(}n\texttt{));}\\ \quad \texttt{goto cmixPendLoop;}\\ \quad m:\ S'\texttt{; cmixExpr(}e'_3\texttt{);}\\ \quad \texttt{cmixGoto(cmixPendinsert(}l\texttt{));}\\ \quad \texttt{goto cmixPendLoop;}\\ \quad n:;\\ \}\end{array}}$$

[label]

$$\frac{\vdash^{stmt} S \Rightarrow S'}{\vdash^{stmt} l:\ S \Rightarrow l:\ S'}$$

[goto] $\vdash^{stmt} \texttt{goto } m \Rightarrow \texttt{goto } m$

[return]

$$\frac{\vdash^{exp} e \Rightarrow e'}{\vdash^{stmt} \texttt{return } e \Rightarrow \texttt{return } e'}$$

$$\frac{\vdash^{exp} e \Rightarrow e'}{\vdash^{stmt} \begin{array}[t]{l}\underline{\texttt{return } e} \Rightarrow\\ \{\\ \quad \texttt{cmixReturn(}e'\texttt{);}\\ \quad \texttt{goto cmixPendLoop;}\\ \}\end{array}}$$

[comp]

$$\frac{\vdash^{stmt} S_i \Rightarrow S'_i}{\vdash^{stmt} \{\ S_1 \ldots S_n\} \Rightarrow \{\ S'_1 \ldots S'_n\}}$$

Figure 31: Transformations of statements (part 2)

[const]	$\vdash^{exp} c \Rightarrow c$	$\frac{\vdash^{exp} e \Rightarrow e'}{\vdash^{exp} \texttt{lift(}e\texttt{)} \Rightarrow \texttt{cmixLift(}e'\texttt{)}}$
[var]	$\vdash^{exp} v \Rightarrow v$	
[struct]	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{exp} e_1.i \Rightarrow e_1'.i}$	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{exp} \underline{e_1.i} \Rightarrow \texttt{cmixStruct(}e_1'\texttt{,}i\texttt{)}}$
[indr]	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{exp} *e_1 \Rightarrow *e_1'}$	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{exp} \underline{*e_1} \Rightarrow \texttt{cmixIndr(}e_1'\texttt{)}}$
[array]	$\frac{\vdash^{exp} e_i \Rightarrow e_i'}{\vdash^{exp} e_1\texttt{[}e_2\texttt{]} \Rightarrow e_1'\texttt{[}e_2'\texttt{]}}$	$\frac{\vdash^{exp} e_i \Rightarrow e_i}{\vdash^{exp} \underline{e_1\texttt{[}e_2\texttt{]}} \Rightarrow \texttt{cmixArray(}e_1'\texttt{,}e_2'\texttt{)}}$
[addr]	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{exp} \&e_1 \Rightarrow \&e_1'}$	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{exp} \underline{\&e_1} \Rightarrow \texttt{cmixAddr(}e_1'\texttt{)}}$
[unary]	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{exp} o\ e_1 \Rightarrow o\ e_1'}$	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{texp} \underline{o\ e_1} \Rightarrow \texttt{cmixUnary(}o\texttt{,}e_1'\texttt{)}}$
[binary]	$\frac{\vdash^{exp} e_i \Rightarrow e_i'}{\vdash^{exp} e_1\ o\ e_2 \Rightarrow e_1'\ o\ e_2'}$	$\frac{\vdash^{exp} e_i \Rightarrow e_i'}{\vdash^{exp} \underline{e_1\ o\ e_2} \Rightarrow \texttt{cmixBinary(}e_1'\texttt{,}o\texttt{,}e_2'\texttt{)}}$
[alloc]	$\vdash^{exp} \texttt{alloc(}T\texttt{)} \Rightarrow \texttt{alloc(}T\texttt{)}$	$\vdash^{exp} \underline{\texttt{alloc(}T\texttt{)}} \Rightarrow \texttt{cmixAlloc(}T\texttt{)}$
[ecall]	$\frac{\vdash^{exp} e_i \Rightarrow e_i'}{\vdash^{exp} f(e_1,\ldots,e_n) \Rightarrow f(e_1',\ldots,e_n')}$	$\frac{\vdash^{exp} e_i \Rightarrow e_i'}{\vdash^{exp} ef\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \Rightarrow \texttt{cmixEcall(}ef\texttt{,}e_1'\texttt{,}\ldots\texttt{,}e_n'\texttt{)}}$
[call]	$\frac{\vdash^{exp} e_i \Rightarrow e_i'}{\vdash^{exp} f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \Rightarrow f\texttt{(}e_1'\texttt{,}\ldots\texttt{,}e_n'\texttt{)}}$	$\frac{\vdash^{exp} e_i \Rightarrow e_i'}{\vdash^{exp} \underline{f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}} \Rightarrow}$ $\texttt{cmixCall(}f\texttt{(}e_1'\texttt{,}\ldots\texttt{,}e_m'\texttt{), }e_{m+1}',\ldots e_n'\texttt{)}$ where e_1,…,e_m static
[pcall]	$\frac{\vdash^{exp} e_i \Rightarrow e_i'}{\vdash^{exp} e_0\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \Rightarrow e_0'\texttt{(}e_1'\texttt{,}\ldots\texttt{,}e_n'\texttt{)}}$	
[preinc]	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{exp} \texttt{++}e_1 \Rightarrow \texttt{++}e_1'}$	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{ext} \underline{\texttt{++}e_1} \Rightarrow \texttt{cmixPre("++",}e_1'\texttt{)}}$
[postinc]	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{exp} e_1\texttt{++} \Rightarrow e_1'\texttt{++}}$	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{ext} \underline{e_1} \Rightarrow \texttt{cmixPost("++",}e_1'\texttt{)}}$
[assign]	$\frac{\vdash^{exp} e_i \Rightarrow e_i'}{\vdash^{exp} e_1\ aop\ e_2 \Rightarrow e_1'\ aop\ e_2'}$	$\frac{\vdash^{exp} e_i \Rightarrow e_i'}{\vdash^{exp} \underline{e_1\ aop\ e_2} \Rightarrow \texttt{cmixAssign(}e_1'\texttt{,"}aop\texttt{",}e_2''\texttt{)}}$
[comma]	$\frac{\vdash^{exp} e_i \Rightarrow e_i'}{\vdash^{exp} e_1, e_2 \Rightarrow e_1', e_2'}$	$\frac{\vdash^{exp} e_i \Rightarrow e_i'}{\vdash^{exp} \underline{e_1, e_2} \Rightarrow \texttt{cmixComma(}e_1'\texttt{,}e_2''\texttt{)}}$
[sizeof]		$\vdash^{exp} \underline{\texttt{sizeof(}T\texttt{)}} \Rightarrow \texttt{cmixSize(}T\texttt{)}$
[cast]	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{exp} \texttt{(}T\texttt{)}e_1 \Rightarrow \texttt{(}T\texttt{)}e_1'}$	$\frac{\vdash^{exp} e_1 \Rightarrow e_1'}{\vdash^{exp} \underline{\texttt{(}T\texttt{)}e_1} \Rightarrow \texttt{cmixCast("}T\texttt{",}e_1'\texttt{)}}$

Figure 32: Transformation rules for expressions

3.10 Memory management

This section is concerned with the management of static stores in a generating extension. Due to the presence of assignments and pointers, this is considerably more complicated than in functional languages. For example, a program may rely on locations, and pointers may point to objects of unknown size.

During execution of the generating extension it is necessary to make copies of the static values, and to restore these at a later time. Furthermore, to implement sharing of residual code, the active state must be compared with previously encountered states.

The memory management is part of the generation-extension library.

3.10.1 Basic requirements

Recall that the part of the store where computations are carried out is called the *active store*. It basically consists of the storage allocated for static objects. A copy of the active store is called a *static store*.

Copies of the active store must be made when a generating function is invoked (in order to determine sharing of the residual function to be generated); after generation of a residual `if` statement (for specialization of the branches), unless residual code can be shared, and after generation of a residual `goto` statement (to enable sharing of the target), unless residual code can be shared.

Copying and comparison of static values during specialization may both be expensive and memory consuming. The basic requirements to the memory management can be listed as follows.

1. The memory management must support comparison of the active store and static stores.
2. The memory management must implement copying of the active store to a new area.
3. The memory management must implement restoring of a static store to be the active store.
4. The memory management must represent static stores “compactly” and enable efficient comparison of values.

These requirements are non-trivial.

During the execution of a generating extension, the static values are allocated on the program stack (in the case of local variables), at statically fixed locations (in the case of global variables), or at the heap. Consider the generating function for a function with a static parameter of type pointer to an integer:

```
Code foo(int *p)
{
   ...
}
```

The parameter 'p' is allocated on the program stack. Suppose that 'foo()' is called, and it must be determined whether it previously has been applied to the same instance of 'p'.

However, where the content of the parameter 'p' easily can be compared with previous values, comparison of the indirection is harder. It does not suffice to compare '*p' since 'p' may point to an array, in which case all entries should be compared. There is, though, no easy way to determine the *size* of the object 'p' points to — the expression 'sizeof(*p)' simply returns the size of ints, not the size of the data structure. Compare for example the calls 'foo(&x)' and 'foo(a)', where 'x' is an integer variable and 'a' is an array.

A specializer overcomes the problem by using a *tagged* representations of static values, but this conflicts with the underlying idea of generating extensions: static values are represented directly. We describe a solution in the next section.

Example 3.38 To see why a *compact* representation is needed, consider the following example. Suppose that 'Pixel image[500][500]' is a global static array.

```
Pixel image[100][100];
int process_image()
{
   if ( dyn-exp )
      ... plot_point(image[x][y]);
   else
      ... save_point(image[x][y]);
}
```

Specialization of the function above causes the array 'image' to be copied twice due to the dynamic `if`. Thus, the generating extension now uses twice the storage as ordinary execution. Clearly, the storage usage may grow exponentially. **End of Example**

In this section we describe a storage model which tries to share copies of static values whenever possible. A further extension is analysis of the program to detect when copying of objects is needless, *e.g.* when an object is dead.

Example 3.39 One may ask why it is necessary to copy and restore the active store when it apparently is not needed in the Flow-chart Mix system [Gomard and Jones 1991a]. The reason is that the flow-chart language does not support locations as first class values, and thus no operations can rely on addresses. The update of the store can henceforth be implemented in a purely functional way (in Scheme, for example), and the copying of the state then happens in a "by-need" manner. **End of Example**

3.10.2 The storage model

We describe a storage model where objects allocated at the same location in the active store may be shared between copies. To each copy of the state we assign a unique *store number*. The store number is employed to differentiate between objects copied due to the processing of different specialization points.

A *store description* is a function from a location and store number to a location: $\mathcal{SD}$: Loc × StNum → Loc. Intuitively, $\mathcal{SD}(l, n) = l'$ means that the copy of the object at location l with store number n is located at address l'.

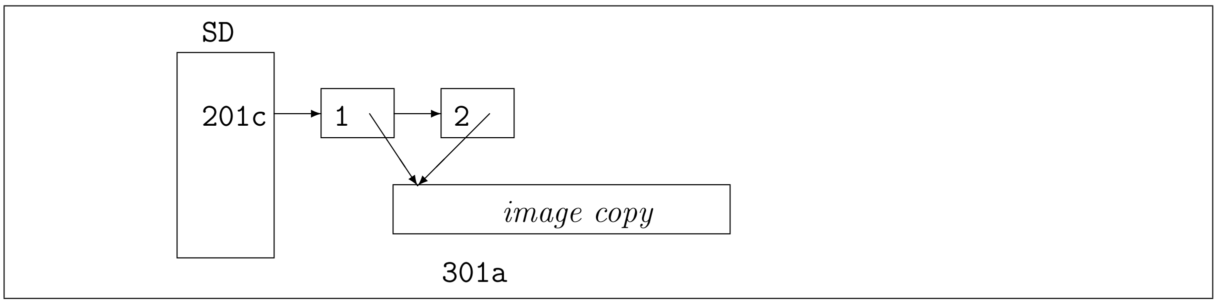

Figure 33: Storage description and copies of static objects

The storage description can for example be implemented as a hash table representing lists of pairs of store numbers and locations, over locations.

Example 3.40 Recall the scenario in Example 3.38 and assume that '`image`' is allocated at location 201c, say. Due to the dynamic conditional, the static state will be copied twice, and the store description will contain the following bindings $[(201c, 1) \mapsto 301a; (201c, 2) \mapsto 301a$. It says that the copies of '`image`' with store numbers 1 and 2 are located at address 301a. See Figure 33. **End of Example**

We assume the following methods on a storage description '`StDesc`':

`StDesc.find` : Loc × StNum → Loc
`StDesc.insert` : Loc × Size × StNum → Loc
`StDesc.restore` : Loc × Size × StNum → Loc

defined as follows.

The method '`find(`l`,`n`)`' returns the location of the n'th copy of the object at location l. The method '`insert(l,s,n)`' copies the object at location l of size s and gives it store number n. The method '`restore(l,s,n)`' restores the n'th copy of the object (originally) from location l of size s.

Example 3.41 Example 3.40 continued. The call '`StDesc.find(201c,1)`' returns the address 301a. The call '`StDesc.restore(201c,100000,1)`' copies the image array to its original position. **End of Example**

3.10.3 State descriptions

To copy an object, its location and size must be known. If the object contains a pointer, the referenced object must also be copied. To determine the size of objects, *state descriptions* are introduced.

A state description is associated with every generating function, and the global state. It records the size and location of static objects. Further, to each object an *object function* is bounded. The object function implements copying, restoring and comparison of object of that type.

Example 3.42 Suppose that '`int n, a[10], *p`' are static variables. A corresponding state description is

```
StateDesc locals[] = { { &x, sizeof(n), &state_int },
                       { &a, sizeof(a), &state_int },
                       { &p, sizeof(p), &state_ptr } };
```

where '`state_int()`' and '`state_ptr()`' are object functions for integer and non-function pointers, respectively. **End of Example**

State descriptions come into existence and disappear together with functions, so they can be represented as automatic variables in generating functions (plus one for the global state). To make the location of the state descriptions currently active known, we use a stack.

Example 3.43 When a (generating) function is entered, its state description is pushed on a StateDesc stack. At function return, it is popped again.

```
Code f(int n)
{
   int a[10], *p;
   StateDesc locals[] = { ... };
   push_state(locals);
   ...
   pop_state();
   return pop_fun();
}
```

The state description stack contains the addresses of all active state descriptions. A state description for a function is active from the function has been invoked till it returns.

Notice how the state description is initialized (by the means of the address operator) to contain the addresses of parameters and local variables. **End of Example**

Now suppose we aim to find the size of the object pointed to by '`p`'. This can be done by scanning of the state descriptions at the state description stack until an object is found that is allocated at the location to which '`p`' points.

Example 3.44 Consider a string constant '`char *str = "Hello"`'. Since string constants cannot be assigned, there is no need to copy these at specialization points, and comparison can be accomplished by pointer equality. **End of Example**

3.10.4 Object copy functions

To copy an object of base type it suffices to know its location and size. The size of a predefined type can be determined via the `sizeof` operator. However, in the case of user defined types, *e.g.* a struct, the copying depends on the definition. For example, if a member is of pointer type, the referenced object must be taken into account. For this an *object function* is associated to each type.

Example 3.45 Let a struct be defined as follows: '`struct S { int x, *p; }`'. To copy an object of that type, the object function first copies the '`x`' and '`p`' members, and next calls the object function for the objects '`p`' points to. **End of Example**

An object function takes a specification flag (copy, compare or restore), the location of the object to handle, and a store number. It returns the size of the object it treats. The algorithm for copying the active store is given below.

Algorithm 3.3 *Copy the state corresponding to '`stat_desc`'.*

```
snum = next_state_number++; /* New unique state number */
/* Copy objects in state description */
for (<loc,size,sfun> in stat_desc) copy_obj(loc,size,snum,sfun);
/* Copy object at loc and size to store number snum */
void copy_obj(void *loc, size_t size, int snum, OFun (*fp)())
{
   StDesc.insert(loc,size,snum);
   for (p = loc; p < loc + size; )
      p += sfun(COPY, p, snum);
   }
}
```

□

The copying works as follows. For each object listed in the state description, the auxiliary function '`copy_obj()`' is called. It copies the data object to the storage description via the function '`insert()`', and calls the object function on each base object.[45]

Example 3.46 The (copy) object function for `int` is simple.

```
/* Copy an int object */
void *state_int(void *loc, int snum)
{
   return sizeof (int);
}
```

The (copy) object function for pointers uses the state description stack to determine the size of the referenced object.

```
/* Copy a pointer object (not function pointer) */
void *state_ptr(void *loc, int snum)
{
   if (*loc != NULL) { /* Not the NULL pointer */
      <loc1,size1,ofun1> = StateDescStack.lookup((void *)*loc,snum);
      copy_obj(loc1,size1,snum,ofun1);
   return sizeof (void *);
}
```

End of Example

[45]For example, in the case of an array, the data object is the entire array, and an object is an entry.

Object functions for base types and pointers can be predefined in the generating-extension library. Object functions for user-defined types can be defined as part of the generating-extension transformation on the basis of the struct definition.

Example 3.47 The state function for the `struct S { int x, *p; }` is given below.

```
void state_S(void *loc, int snum)
{
   if ((loc = (struct S *)loc->p) != NULL) {
      <loc1,size1,ofun1> = StateDescStack.lookup(loc, snum);
      copy_obj(loc1,size1,snum,ofun1);
   }
   return sizeof (struct S);
}
```

Notice that if a pointer references a member of a struct, the whole object — and not just the member — is copied. This introduces a slight overhead, but does not matter in practice. **End of Example**

A note. The C standard allows a pointer to point one past an array but not such a pointer to be dereferenced. Consider the following situation. Given definitions '`int *p, a[10], b[10]`' where '`p`' points to '`a + 10`'. Even though a program cannot rely on '`b`' being allocated in continuation of '`a`', it is often the case. This confuses the memory management: does '`p`' actually points to '`a`' or to '`b`'? To be safe, both are copied.

The algorithms for restoring and comparing objects are similar to Algorithm 3.3, and hence omitted.

The memory management may seem expensive but it is not unacceptable in practice. Recall that side-effects normally are suspended, with the consequence that only local and global objects needs to be copied. Heap-allocated data structures are seldom copied.

Example 3.48 The memory usage can be reduced dramatically due to the following observation. Sharable functions commit no side-effects It is therefore not necessary to copy non-local objects, since they cannot be assigned. The state description can be employed to determined whether an object is local or non-local. **End of Example**

3.10.5 The seenB4 predicate

The '`seenB4()`' predicate shall return true if a function's static parameters and the global variables match a previous invocation, and the function is specified to be sharable. This means that a copy of the values to which a function was specialized must be associated with each residual function.[46]

A minor complication must be taken into account. The parameters of a function are not necessarily allocated at the same location at the program stack every time the function is called. It is easy to program around this problem, however.

[46]There is no need to save a copy for functions that cannot be shared.

3.10.6 Improved sharing of code

The development so far is sufficient for the handling of the static stores in a generating extension. For example, a dynamic `goto` is transformed into the statements

```
cmixGoto(cmixPendinsert(label)); goto cmixPendLoop;
```

where '`cmixPendinsert()`' uses the '`StDesc.insert()`' function to copy the active store. However, to increase the amount of sharing, specialization can be restricted to *live variables.*

Example 3.49 Assume that `exp` is a dynamic expression, and that the `goto` statements are specialized.

```
if (exp) { x = 1; goto l; }  else { x = -1; goto l; }
l: x = 13;
```

At program point '`l`', '`x`' is dead, but it will still be specialized twice since '`x`' differs at the two `goto` statements. By specializing to live variables only, the program point '`l`' will be shared. This observation was originally made in the setting of a flow-chart language [Gomard and Jones 1991a]. **End of Example**

To accomplish this, we extend the function `cmixPendinsert()` to take a bit-string:

```
cmixGoto(cmixPendinsert(label, "1001")); cmix cmixPendLoop;
```

where '`"1001"`' indicates live variables.

Chapter 6 develops an *in-use variable analysis* that computes the desired information.

Example 3.50 The following function increments a pointer.

```
/* inc_ptr: increment p by one */
int *int_ptr(int *p)
{
   return p + 1;
}
```

Suppose that '`inc_ptr()`' is specialized with respect to an instance of '`p`'. Normally, both the content and the indirection of '`p`' will be employed to determine sharing of the residual version of '`int_ptr()`'. However, since the indirection is not used in the body, it should not be taken into account. The in-use analysis developed in Chapter 6 recognizes such situations. **End of Example**

3.10.7 Heap-allocated memory

Recall that runtime memory allocation (at specialization time) is performed by the means of the mix-function '`alloc()`'. Calls to '`malloc()`' and its cognates are suspended.

To allow tracing of pointers pointing to heap-allocated objects, '`alloc()`' must maintain a state description of the heap. Since heap-allocated data structures tend to be rather large, copying and comparing of heap-allocated data structures should be kept to a minimum. We have currently no automatic strategy for when side-effects on heap allocated objects should be allowed.

Example 3.51 Several functions in the standard library allocate memory, for instance '`strdup()`'. In some applications it is undesirable to suspend calls to memory allocating functions. For that reason, *C-Mix* contains definitions of library functions which use the '`alloc()`' function for storage allocation. **End of Example**

A final remark. Data structures may be circular. This further complicates both copying, comparing and restoring, since the objects "seen" so far must be accumulated, to prevent circular processing.

3.11 Code generation

This section briefly reconsiders code generating functions implemented by the generating-extension library, and presents two improvements. The former gives "nicer" looking residual programs, the latter may actually improve specialization.

3.11.1 Algebraic reductions

Consider a dynamic expression such as 'e `+ 0`', which is transformed into the code generating calls '`cmixBinary(`$\overline{e}$`,"+",cmixInt(0))`'. (To make the example non-trivial, suppose the 0 is the result of a complicated static evaluation.) Obviously, by implementing the *algebraic reduction* $e + 0 = e$ into '`cmixBinary`', the plus operator can be eliminated from the residual program.[47]

This does not in general improve the efficiency of residual programs — a (good) optimizing compiler will do the same kind of transformations — but yields "nicer" residual programs. Algebraic reduction serves another purpose, though.

Example 3.52 Two useful algebraic rules are $0\&\&e = 0$ and $1||e = 1$ since they often appear in the test of *e.g.* an `if`. Notice that the reductions are safe, since the semantics of C guarantees that e never will be evaluated. **End of Example**

[47]Beware: the rule $e * 0 = 0$ discards a possibly non-terminating expression!

3.11.2 Deciding dynamic tests

Consider again the '`strstr()`' example from Section 3.2, and recall that the last conjunct in '`--n >= needle && --h >= haystack`' was commented out to prevent the `do-while` loop to become dynamic. This section shows an extension that sidesteps this problem.

Consider the specialization of a dynamic `if` statement. As presented in the previous sections, both branches will always get specialized. However, it may be the case that the test actually can be decided, even though it is dynamic.[48] In that case the is no need to specialize both branches: the transition can be compressed. This is accomplished as follows.

An '`if (`e`)` S_1 `else` S_2' statement is transformed into the following code:

```
                              if_test = e';
if (e)                        switch (cmixTest(if_test)) {
   S1                           case 1:  /* test is true */  goto m;
else            =gegen=>        case 0:  /* test is false */ goto n;
   S2                           default: /* test is undecidable - specialize */
                                  cmixIf(if_test,cmixPendinsert(m),cmixPendinsert(n));
                                  goto cmixPendLoop;
                              }
                            m: S1'; goto l;
                            n: S2'; goto l;
                            l:;
```

The function '`cmixTest()`' returns 1 if the argument is the representation of a non-zero constant, 0, if it is the zero constant, and -1 otherwise.

Example 3.53 Applying on-line test of `if` expression allows the '`strstr()`' function to be specialized without any rewriting. **End of Example**

The improvement implements a "mix-line" strategy for determination of a value's binding time [Jones *et al.* 1993, Chapter 7]. It is also a first step toward positive context propagation, which we consider in Chapter 10.

Example 3.54 This improvement has been implemented in *C-Mix* by a slight change of '`cmixIf()`'. If the test expression is a constant, it generates a jump to the corresponding branch. **End of Example**

3.12 Domain of re-use and sharing

In the previous section we defined the predicate '`seenB4()`' to return true on exact match. This is often an over-strict requirement that prevents sharing of semantically equivalent functions. This section introduces *domain of re-use* and lists sufficient criteria for sharing of a specialized functions. Although the discussion is centered around function specialization, the techniques carry over to program point specialization.

[48] Recall: dynamic = *possibly* unknown.

3.12.1 Domain of specialization

Functions are specialized with respect to values of base type, struct type, pointers, and composed types.

Definition 3.3 *The set of types of values to which functions are specialized is called the* domain of specialization.[49] □

A necessary requirement for a type to be in the domain of specialization is that an equality operator is defined over the type. For example, the higher-order partial evaluators Similix [Bondorf 1990] and Schism [Consel 1993b] use representations of closures to recognize equality of functions. All expressable values in C are comparable.

3.12.2 Domain of re-use

In this section we for convenience assume that global variables are "part" of a function's arguments. So far we have implicitly assumed that a residual function is shared only when the static arguments match. In many cases, however, this is an excessively strict criteria. Consider for example the following function,

```
/* choose: return x or y depending on p */
int choose(int p, int x, int y)
{
   return p? x : y;
}
```

and assume it is specialized with respect to a static 'p'. By strict comparison of 'p' to determine sharing, we get a specialized version for every value of 'p'. Clearly, it suffices to construct two versions, one for 'p' being non-zero, and one for 'p' being zero.

Example 3.55 We can remedy the situation by a small trick.

```
/* Generating extension for choose */
Code choose(int p)
{
   p = p != 0;
   if (seenB4()) return  name_of_function;
   /* specialize */
   ...
}
```

The trick is to "normalize" 'p' before it is compared against previous values of 'p'. Naturally, this requires insight into the function. **End of Example**

Definition 3.4 *Let f be a function taking n arguments, and assume that the first $m \leq n$ are static. Let f' be a specialized version of f with respect an instance $v_1, \ldots, v_m$. The* domain of re-use*, $DOR(f')$ for function f' is defined to be a largest set of values in $V_1 \times \cdots \times V_m$ such that*

[49]Ruf uses this term equivalently to our domain of re-use [Ruf and Weise 1991]. We believe that our terminology is more intuitive and suggestive.

$$f(v_1, \ldots, v_n) = v \vee f(v_1, \ldots, v_n) = \bot \Leftrightarrow f'(v_{m+1}, \ldots, v_n) = v$$

for all $(v_1, \ldots, v_m) \in V_1 \times \cdots \times V_m$ *and all* $v_{m+1}, \ldots, v_n$ *in the domain of* f. □

The domain of re-use is the set of static values where a particular residual function computes the same result for all dynamic arguments.

Example 3.56 Suppose that '`choose()`' is specialized with respect to '`p`' being 1, resulting in '`choose_p()`'. We have DOR(`choose_p()`) $= \{n \mid n \neq 0\}$. Notice that the domain of re-use is defined with respect to a residual function. **End of Example**

When a residual call is met during specialization, and the static arguments are in the domain of re-use of a residual function, it can be shared.

In general, the domain of re-use is uncomputable, so an approximation must do.

3.12.3 Live variables and domain of re-use

In a previous section we claimed that specialization with respect to live variables gives better sharing.

Example 3.57 Consider the following function where the argument '`x`' does not contribute to the value returned.

```
int foo(int x, int y)
{
   return y;
}
```

Suppose that '`x`' is static. We have: DOR(`foo`$'$) $= I\!N$. If specialization performed with respect to live variables only, the residual version will be shared for all values of '`x`', as desired. **End of Example**

The use of liveness information only yields a rude approximation of the domain of re-use. The problem is that liveness information does not take value dependencies into account. For example, it fails to see that the outcome of '`x > 0`' only can be 1 or 0.

Example 3.58 Consider again the '`inc_ptr()`' function. A residual version can be shared between calls with the same value of '`p`' — the content of the indirection is immaterial. Classical live-variable analysis does not recognize that '`*p`' is dead. The in-use analysis we develop in Chapter 6 does. **End of Example**

3.13 Specialization, sharing and unfolding

A (dynamic) function in a subject program can be handled in one of the following three ways by specialization: *i*) be specialized and residual versions possibly shared; *ii*) be specialized unconditionally such that no residual versions are shared, or *iii*) be unfolded into the caller. As we have seen previously, the choice of treatment interacts with the binding time classification of side-effects, and may thus influence greatly the result of specialization.

At the time of writing we have not automated a strategy in the *C-Mix* system. Functions are specialized and shared, unless otherwise specified by the user.[50] In this section we outline a possible strategy. It should be noted, however, that is it is easy to find examples where the strategy fails.

3.13.1 Sharing and specialization

Recall that to share residual versions of a function, all side-effects in the function must be suspended, the reason being that static variables otherwise may need updating after sharing (at specialization time). If sharing is abandoned, side-effects can be allowed, but not side-effects under dynamic control.

Example 3.59 Observe that allowing a function to accomplish a static side-effect on *e.g.* a global variable implies that callers of the function also accomplish side-effects. Thus, side-effects "back-propagate". **End of Example**

Preferable residual functions should always be shared when feasible (and possible), to reduce the size of the residual program and the risk of non-termination. In some cases, however, static side-effects are necessary. For example, in a function initializing global data structures, it seems permissible to allow an "init" function to side-effect since it only will be called once, and hence no need for sharing actually exists.

Necessary requirements for unconditional specialization of a function are listed below.

- The function is non-recursive, and
- is a terminal function (*i.e.*calls no other functions), and
- it is "close" to the '`main()`' function in the program's invocations graph.

Recall that functions (possibly) referred to via a function pointer shall contain no static side-effects. The requirements are justified as follows.

If a function is recursive folding is necessary to prevent infinite function specialization. A function is called a terminal function if it calls no other functions. This condition is included to reduce code duplication. Finally, the latter condition ensures that side-effects do not back-propagate to "too many" functions in the case of a long invocation chain (*e.g.* '`main()`' calls '`foo()`' that calls '`bar()`' that calls '`baz()`' which contain a side-effect, causing '`bar()`' and '`foo()`' to be non-sharable).

[50]The *C-Mix* system supports the specifiers 'specialize' and 'unfold'.

Example 3.60 Runtime memory allocation by the means of '`alloc()`' performs a side-effect on the heap. By the strategy above, runtime allocation performed by (dynamic) recursive functions will be deferred to runtime. **End of Example**

Chapter 6 a side-effect analysis to detect side-effects possibly under dynamic control. Both are needed to implement the sharing strategy.

3.13.2 Unfolding strategy

An unfolding strategy must prevent infinite unfolding and avoid extensive code duplication. In the original Mix, the call unfolding was based on detection of inductive arguments. Only functions with a static, inductive parameter were unfolded [Sestoft 1988]. The Scheme0 partial evaluator unfolds all calls not appearing in the branch of a dynamic `if` [Jones *et al.* 1993]. The Similix system has a strategy where dynamic loops are broken by insertion of so-called *sp-functions* (specialization functions). All calls to functions but sp-functions are unfolded [Bondorf 1990].

At the time of writing we have implemented no unfolding strategy into *C-Mix*. Functions are only unfolded during specialization if specified by the user via an '`unfold`' specifier. There are three reasons to this. First, unfolding complicates the handling of the pending list and the memory management, introducing an overhead in generating extensions. Secondly, to prevent infinite unfold a strategy must be so conservative that post-unfolding is needed anyway. Finally, it is undesirable to unfolding "big" functions unless they are called only once. However, neither the size nor the number of times a residual function is called can be determined during specialization — first after specialization. For these reasons it seems practical to defer all unfolding to the post-processing stage, there more advanced decision strategies for unfolding can be employed [Hwu and Chang 1989].

Example 3.61 Inlining of function '`foo()`' is clearly undesirable unless it is "small".

```
for (n = 0; n < 100; n++) foo(n);
```

Notice that it may be the case that the number of (static) call-sites to a function is one, and unfolding is undesirable anyway. **End of Example**

Necessary requirements for a function to be unfolded during specialization are:

- It is non-recursive, and
- cannot be called via a function pointer.

Clearly, unfolding of recursive function loops. The equivalent to Similix's use of sp-functions could be adapted to break loops. Functions called via function pointers must always be specialized, and since the unfolding technique is function based, and not call-site based, indirectly called functions cannot be unfolded.

Currently we believe that as much function unfolding as possible should be done by post-processing, but since on-line unfolding may improve the binding time classification, it cannot be abandoned completely.

3.14 Imperfect termination

Generating extensions suffer from the same termination problems as ordinary program specializers: they may fail to terminate on programs that normally stop. One reason is that an extension is *hyperstrict*; it "executes" both branches of a dynamic `if` whereas normal execution at most executes one. Non-termination may also be due to infinite transition compression (in the case of static loops), or simply because the subject program loops on the given static input independently of dynamic data. A program specializer is often accepted to loop in the latter situation, but should ideally terminate in the former cases.

Example 3.62 A generating extension for the following lines loops.

```
int n;
for (n = 100; dyn-exp < n ; n--) ;
```

To hinder non-termination, the variable `n` must be generalized. **End of Example**

Jones has formulated a strict terminating binding-time analysis for a first-order functional language based on inductive arguments [Jones 1988]. Holst has refined the termination analysis, and proved its correctness [Holst 1991]. The prevailing idea in these analyses is that if a static value grows along a transition, another static value must decrease. This can be computed by a dependency analysis which approximates the dependencies between variables at each program point.

These methods seem not suitable for partial evaluation of C. The analyses rely on downward closed domains, and can for example not handle the example above. We have investigated some methods but have at the time of writing no solution.

3.15 Related work

The work reported in this chapter continues the endeavor on automatic specialization and optimization of imperative programs. For a more comprehensive description we refer to [Jones *et al.* 1993].

3.15.1 Partial evaluation: the beginning

Partial evaluation is the constructive realization of Kleenee's S-m-n theorem. Where the proof for Klenee's theorem mainly is concerned with *existence*, the aim of partial evaluation, and more general program specialization, is *efficiency*. Futamura formulated the (first two) Futamura projections in 1977, but no practical experiments were conducted [Futamura 1971].

The first experiments began with the work of Beckman *et al.* on the RedFun partial evaluator [Beckman *et al.* 1976]. At the same time Turchin described *super-compilation*, a technique that subsumes partial evaluation [Turchin 1979]. Ershov studied partial evaluation for small imperative languages [Ershov 1977].

In 1985, Jones, Sestoft and Søndergaard developed and implemented the first self-applicable partial evaluator for a subset of Lisp [Jones *et al.* 1989]. Following the same basic principles, Bondorf and Danvy developed the Similix partial evaluator than handles a substantial, higher-order subset of the Scheme programming language [Bondorf 1990], and Consel developed Schism for a similar language [Consel 1993b].

3.15.2 Partial evaluation for imperative languages

Partial evaluation (or mixed computation) was originally formulated for small imperative languages by Ershov and Bulyonkov [Ershov 1977,Bulyonkov and Ershov 1988]. Gomard and Jones developed an off-line self-applicable partial evaluator for a flow-chart language with Lisp S-expressions as the sole data structure [Gomard and Jones 1991a, Jones *et al.* 1993].

The author developed and implemented an off-line self-applicable partial evaluator for a subset of the C programming language including functions, global variables, arrays, and to some extent pointers [Andersen 1991,Andersen 1992,Andersen 1993b]. The specializer kernel was successfully self-applied to generate compilers and a compiler generator.

Meyer has described on-line partial evaluator for parts of Pascal, excluding heap-allocation and pointers [Meyer 1992]. Nirkhe and Pugh have applied a partial evaluator for a small C-like language to hard real-time systems [Nirkhe and Pugh 1992]. Baier *et al.* have developed a partial evaluator for a subset of Fortran [Baier *et al.* 1994]. Blazy and Facon have taken a somewhat different approach to partial evaluation of Fortran [Blazy and Facon 1993]. The aim is to "understand" programs by specialization to the input of interest.

3.15.3 Generating extension generators

A main motivation in favor of generating extension as opposed to traditional specialization, is preservation of programs' semantics. This issue is, however, also prevailing in optimizing compilers that attempt compile-time evaluation of expressions. Thompson suggested that compile-time expressions are *executed* by the underlying hardware at compile-time, to avoid erroneous interpretations [Aho *et al.* 1986] — exactly as generating extensions execute static constructs.

Beckman *el al.* have developed a compiler generator for a subset of the Lisp language [Beckman *et al.* 1976]. The aim of the RedCompiler was to optimize compilers, but the project was ahead of its time: a precise semantics foundation was lacking.

Ghezzi *et al.* study program simplification, which is a generalization of mixed computation [Ghezzi *et al.* 1985]. During symbolic evaluation of a program, constraints about program predicates are collected and exploited by a expression reducer (theorem prover). For example, in the then-branch of an `if`, the predicate is recorded to be true. Coen-Porisini *et al.* have extended the methods further and incorporated them into Ada [Coen-Porisini *et al.* 1991].

Pagan describes hand-conversions of programs into their generating extension in a Pascal framework [Pagan 1990]. Mainly to overcome the problem with doubly encoded programs when self-applying specializers for strongly-typed languages, Holst and Launchbury suggested hand-writing of the compiler generator '`cogen`' [Holst and Launchbury 1991]. A small ML-like language without side-effects was used as example language.

Birkedal and Welinder have developed a compiler generator for the Core Standard ML language [Birkedal and Welinder 1993]. Their system does not specialize functions with respect to higher-order values nor references and exceptions. However, specialization of pattern matching seems to yield significant speedups.

Bondorf and Dussart have implemented a CPS-cogen for an untyped lambda calculus [Bondorf and Dussart 1994]. The CPS-style allows, at specialization time, that static evaluations in dynamic contexts are carried out.

A slightly different approach, deferred compilation, have been investigated by Leone and Lee [Leone and Lee 1993]. The aim is to defer code generation to runtime, where input is available. Function calls are compiled into calls to generating functions that produce specialized versions of the function (in machine code). After the generating, a jump to the to code is performed. Preliminary experiments show a speedup of more than 20 %.

3.15.4 Other transformations for imperative languages

Most program transformation techniques are formulated for functional or logic programming languages, *e.g.* fold/unfold transformations [Burstall and Darlington 1977], supercompilation [Turchin 1986], and finite set differencing [Cai and Paige 1993]. A fundamental reason for this is the complicated semantics of imperative languages, *e.g.* side-effects and pointers.

In Chapter 10 we study driving of C, a strictly more powerful transformation than partial evaluation.

3.16 Future work and conclusion

Although we have described transformation of the full Ansi C programming language, there are much room for improvements.

3.16.1 Further work

Structures. In our experience, structs are the most complicated data structure to specialize. Future work include methods for passing of partially-static structs between functions, and binding-time improving transformations.

Dynamic memory allocation. In this thesis we have assumed that the user identifies runtime memory allocations, and replaces them by '`alloc()`' calls. Preferably, of course, the system should do this automatically. More generally, the treatment of heap-allocated data structures should be improved.

A fruitful way to proceed may be by inferring of the *structure* of heap-allocated data structures. For example, knowing that a pointer points to a singly-linked list may allow more a static binding-time separation. Manual specification of the data structures have been proposed by Hendren [Hendren and Hummel 1992], and Klarlund and Schwartzbach [Klarlund and Schwartzbach 1993]. We suspect that the most commonly used data structures, *e.g.* single and double linked list, binary trees, can be inferred automatically, and the involved pointers annotated.

Memory mamagement. Sometimes a generating extension's memory usage explodes. The copying of the static part of the store at specialization points is problematic: it may easily exhaust the memory, and slows specialization down.

A *last-use* analysis may be employed to free allocated memory when it is certain that the saved state is not needed anymore. For example, if it is known that a generating function cannot be called again, the call signature can be deleted. Further, if an analysis can detect that "similar" calls to a function never will occur, and thus no residual functions will be shared, the call signature can be discarded. Various techniques have been suggested by Malmkjær, and should be investigated further [Malmkjær 1993].

Termination. The generating extensions suffer from (embarrassingly) imperfect termination properties. However, none of the existing methods for assuring termination seem to be suitable for imperative languages. Methods for improving termination are needed. These must probably be coupled with program transformations to avoid excessive conservative binding time annotations.

Approximation of DOR. Residual code sharing the the present version of *C-Mix* is poor. An analysis for approximation of the domain of re-use could improve upon this. A possible way may be by approximation of parameter dependency on a function's results via a backward analysis.

Unfolding and sharing strategies. Currently, we have no good on-line unfolding strategy nor strategy for unconditional specialization of functions. An analysis based on a program's invocation graph is possible.

3.16.2 Conclusion

We have developed and described a generating-extension generator for the Ansi C programming language. The generator takes a binding-time annotated program and converts it into a generating extension, that produces specialized versions of the original program.

A major concern in automatic program transformation is preservation of semantics. We have argued that specialization by the means of generating extensions is superior to traditional specialization via symbolic evaluation. Furthermore, we have described several other advantages.

We studied specialization of pointers, structures, arrays and runtime memory allocation, and gave the generating-extension transformation of those constructs. Furthermore, we have developed a generating-extension library implementing memory management and code generation.

Finally, we have considered propagation of partially-known information via algebraic reductions, and reduction of `if`.

We have implemented the generating-extension generator into the *C-Mix* system. Chapter 9 reports experimental results.

Chapter 4

Pointer Analysis

We develop an efficient, inter-procedural pointer analysis for the C programming language. The analysis approximates for every variable of pointer type the set of objects it may point to during program execution. This information can be used to improve the accuracy of other analyses.

The C language is considerably harder to analyze than for example Fortran and Pascal. Pointers are allowed to point to both stack and heap allocated objects; the address operator can be employed to compute the address of an object with an lvalue; type casts enable pointers to change type; pointers can point to members of structs; and pointers to functions can be defined.

Traditional pointer analysis is equivalent to alias analysis. For example, after an assignment '`p = &x`', '`*p`' is aliased with '`x`', as denoted by the alias pair $\langle *p, x\rangle$. In this chapter we take another approach. For an object of pointer type, the set of objects the pointer *may point to* is approximated. For example, if in the case of the assignments '`p = &x; p = &y`', the result of the analysis will be a map $[p \mapsto \{x, y\}]$. This is a more economical representation that requires less storage, and is suitable for many analyses.

We specify the analysis by the means of a non-standard *type inference system*, which is related to the standard semantics. From the specification, a *constraint-based* formulation is derived and an efficient inference algorithm developed. The use of non-standard type inference provides a clean separation between specification and implementation, and gives a considerably simpler analysis than previously reported in the literature.

This chapter also presents a technique for *inter-procedural* constraint-based program analysis. Often, context-sensitive analysis of functions is achieved by copying of constraints. This increases the number of constraints exponentially, and slows down the solving. We present a method where constraints over *vectors* of pointer types are solved. This way, only a few more constraint are generated than in the intra-procedural case.

Pointer analysis is employed in the *C-Mix* system to determine side-effects, which is then used by binding-time analysis.

4.1 Introduction

When the lvalue of two objects coincides the objects are said to be *aliased*. An alias is for instance introduced when a pointer to a global variable is created by the means of the address operator. The aim of alias analysis is to approximate the set of aliases at runtime. In this chapter we present a related but somewhat different pointer analysis for the C programming language. For every pointer variable it computes the set of abstract locations the pointer may point to.

In languages with pointers and/or call-by-reference parameters, alias analysis is the core part of most other data flow analyses. For example, live-variable analysis of an expression '`*p = 13`' must make worst-case assumptions without pointer information: '`p`' may reference all (visible) objects, which then subsequently must be marked "live". Clearly, this renders live-variable analysis nearly useless. On the other hand, if it is known that only the aliases $\{\langle *p, x\rangle, \langle *p, y\rangle\}$ are possible, only '`x`' and '`y`' need to be marked "live".

Traditionally, aliases are represented as an equivalence relation over abstract locations [Aho *et al.* 1986]. For example, the alias introduced due to the expression '`p = &x`' is represented by the *alias set* $\{\langle *p, x\rangle\}$. Suppose that the expressions '`q = &p; *q = &y`' are added to the program. The alias set then becomes $\{\langle *p, x\rangle, \langle *q, p\rangle, \langle **q, x\rangle, \langle **q, y\rangle\}$, where the latter aliases are *induced* aliases. Apparently, the size of an alias set may evolve rather quickly in a language with multi-level pointers such as C. Some experimental evidence: Landi's alias analysis reports more than 2,000,000 program-point specific aliases in a 3,000 line program [Landi 1992a].

Moreover, alias sets seem excessively general for many applications. What needed is an answer to "which objects may this pointer point to"? The analysis of this chapter answer this question.

4.1.1 What makes C harder to analyze?

The literature contains a substantial amount of work on alias analysis of Fortran-like languages, see Section 4.11. However, the C programming language is considerably more difficult to analyze; some reasons for this include: multi-level pointers and the address operator '&', structs and unions, runtime memory allocations, type casts, function pointers, and separate compilation.

As an example, consider an assignment '`*q = &y`' which adds a point-to relation to p (assuming '`q`' points to '`p`') even though '`p`' is not syntactically present in the expression. With only single-level pointers, the variable to be updated is syntactically present in the expression.[1] Further, in C it is possible to have pointers to both heap and stack allocated objects, as opposed to Pascal that abandon the latter. We shall mainly be concerned with analysis of pointers to stack allocated objects, due to our specific application.

A special characteristic of the C language is that implementation-defined features are supported by the Standard. An example of this is cast of integral values to pointers.[2]

[1]It can easily be shown that call-by-reference and single-level pointers can simulate multi-level pointers.

[2]Recall that programs relying on implementation-defined features are non-strictly conforming.

Suppose that '`long table[]`' is an array of addresses. A cast '`q = (int **)table[1]`' renders '`q`' to be implementation-defined, and accordingly worst-case assumptions must be made in the case of '`*q = 2`'.

4.1.2 Points-to analysis

For every object of pointer type we determine a *safe* approximation to the set of locations the pointer *may* contain during program execution, for all possible input. A special case is function pointers. The result of the analysis is the set of functions the pointer may invoke.

Example 4.1 We represent point-to information as a map from program variables to sets of object "names". Consider the following program.

```
int main(void)
{
   int x, y, *p, **q, (*fp)(char *, char *);
   p = &x;
   q = &p;
   *q = &y;
   fp = &strcmp;
}
```

A safe point-to map is

$$[p \mapsto \{x, y\}, q \mapsto \{p\}, fp \mapsto \{strcmp\}]$$

and it is also a *minimal* map. **End of Example**

A point-to relation can be classified *static* or *dynamic* depending on its creation. In the case of an array '`int a[10]`', the name '`a`' statically points to the object '`a[]`' representing the content of the array.[3] Moreover, a pointer to a struct points, when suitable converted, to the initial member [ISO 1990]. Accurate static point-to information can be collected during a single pass of the program.

Point-to relations created during program execution are called *dynamic*. Examples include '`p = &x`', that creates a point-to relation between '`p`' and '`x`'; an '`alloc()`' call that returns a pointer to an object, and '`strdup()`' that returns a pointer to a string. More general, *value setting* functions may create a dynamic point-to relation.

Example 4.2 A point-to analysis of the following program

```
char *compare(int first, char *s, char c)
{
   char (*fp)(char *, char);
   fp = first? &strchr : &strrchr;
   return (*fp)(s,c);
}
```

will reveal $[fp \mapsto \{strchr, strrchr\}]$. **End of Example**

It is easy to see that a point-to map carries the same information as an alias set, but it is a more compact representation.

[3]We treat arrays as aggregates.

4.1.3 Set-based pointer analysis

In this chapter we develop a flow-insensitive *set-based point-to analysis* implemented via *constraint solving.* A set-based analysis consists of two parts: a specification and an inference algorithm.

The specification describes the *safety* of a pointer approximation. We present a set of inference rules such that a pointer abstraction map fulfills the rules only if the map is safe. This gives an algorithm-independent characterization of the problem.

Next, we present a *constraint-based* characterization of the specification, and give a constraint-solving algorithm. The constraint-based analysis works in two phases. First, a *constraint system* is generated, capturing dependencies between pointers and abstract locations. Next, a *solution* to the constraints is found via an iterative solving procedure.

Example 4.3 Consider again the program fragment in Example 4.1. Writing T_p for the abstraction of 'p', the following constraint system could be generated:

$$\{T_p \supseteq \{x\}, T_q \supseteq \{p\}, *T_q \supseteq \{y\}, T_{fp} \supseteq \{strcmp\}\}$$

with the interpretation of the constraint $*T_q \supseteq \{y\}$: "the objects 'q' may point to contain y". **End of Example**

Constraint-based analysis resembles classical data-flow analysis, but has a stronger semantical foundation. We shall borrow techniques for iterative data-flow analysis to solve constraint systems with finite solutions [Kildall 1973].

4.1.4 Overview of the chapter

This chapter develops a flow-insensitive, context-sensitive constraint-based point-to analysis for the C programming language, and is structured as follows.

In Section 4.2 we discuss various degrees of accuracy a value-flow analysis can implement: intra- and inter-procedural analysis, and flow-sensitive versus flow-insensitive analysis. Section 4.3 considers some aspects of pointer analysis of C.

Section 4.4 specifies a sticky, flow-insensitive pointer analysis for C, and defines the notion of *safety.* In Section 4.5 we give a constraint-based characterization of the problem, and prove its correctness.

Section 4.6 extends the analysis into a context-sensitive inter-procedural analysis. A sticky analysis merges all calls to a function, resulting in loss of precision. We present a technique for context-sensitive constraint-based analysis based on static-call graphs.

Section 4.7 presents a constraint-solving algorithm. In Section 4.8 we discuss algorithmic aspects with emphasis on efficiency, and Section 4.9 documents the usefulness of the analysis by providing some benchmarks from an existing implementation.

Flow-sensitive analyses are more precise than flow-insensitive analyses. In Section 4.10 we investigate program-point, constraint-based pointer analysis of C. We show why multi-level pointers render this kind of analysis difficult.

Finally, Section 4.11 describe related work, and Section 4.12 presents topics for future work and concludes.

4.2 Pointer analysis: accuracy and efficiency

The precision of a value-flow analysis can roughly be characterized by two properties: flow-sensitivity and whether it is inter-procedural vs. intra-procedural. Improved accuracy normally implies less efficiency and more storage usage. In this section we discuss the various degrees of accuracy and their relevance with respect to C programs.

4.2.1 Flow-insensitive versus flow-sensitive analysis

A data-flow analysis that takes control-flow into account is called *flow-sensitive.* Otherwise it is *flow-insensitive.* The difference between the two is most conspicuous by the treatment of `if` statements. Consider the following lines of code.

```
int x, y, *p;
if ( test ) p = &x; else p = &y;
```

A flow-sensitive analysis records that in the branches, '`p`' is assigned the address of '`x`' and '`y`', respectively. After the branch, the information is merged and '`p`' is mapped to both '`x`' and '`y`'. The discrimination between the branches is important if they for instance contain function calls '`foo(p)`' and '`bar(p)`', respectively.

A flow-insensitive analysis summarizes the pointer usage and states that '`p`' may point to '`x`' and '`y`' in both branches. In this case, spurious point-to information would be propagated to '`foo()`' and '`bar()`'.

The notion of flow-insensitive and flow-sensitive analysis is intimately related with the notion of *program-point specific* versus *summary* analysis. An analysis is *program-point specific* is if it computes point-to information for each program point.[4] An analysis that maintains a summary for each variable, valid for *all* program points of the function (or a program, in the case of a global variable), is termed a *summary* analysis. Flow-sensitive analyses must inevitably be program-point specific.

Flow-sensitive versus in-sensitive analysis is a trade off between accuracy and efficiency: a flow-sensitive analysis is more precise, but uses more space and is slower.

Example 4.4 Flow-insensitive and flow-sensitive analysis.

```
/* Flow-insensitive */         /* Flow-sensitive */
int main(void)                 int main(void)
{                              {
   int x, y, *p;                  int x, y, *p;
   p = &x;                        p = &x;
   /* p ↦ { x,y } * /             /* p ↦ {x} */
   foo(p);                        foo(p);
   p = &y;                        p = &y;
   /* p ↦ { x,y } */              /* p ↦ {y} */
}                              }
```

[4]The analysis does not necessarily have to compute the complete set of pointer variable bindings; only at "interesting" program points.

Notice that in the flow-insensitive case, the spurious point-to information $p \mapsto \{y\}$ is propagated into the function '`foo()`'. **End of Example**

We focus on flow-insensitive (summary) pointer analysis for the following reasons. First, in our experience, most C programs consist of many small functions.[5] Thus, the extra approximation introduced by summarizing all program points appears to be of minor importance. Secondly, program-point specific analyses may use an unacceptable amount of storage. This, pragmatic argument matters when large programs are analyzed. Thirdly, our application of the analysis does not accommodate program-point specific information, *e.g.* the binding-time analysis is program-point insensitive. Thus, flow-sensitive pointer analysis will not improve upon binding-time separation (modulo the propagation of spurious information — which we believe to be negligible).

We investigate program-point specific pointer analysis in Section 4.10.

4.2.2 Poor man's program-point analysis

By a simple transformation it is possible to recover some of the accuracy of a program-point specific analysis, without actually collecting information at each program point.

Let an assignment $e_1 = e_2$, where e_1 is a variable and e_2 is independent from pointer variables, be called an *initialization assignment.* The idea is to rename pointer variables when they are initialized.

Example 4.5 Poor man's flow-sensitive analysis of Example 4.4. The variable '`p`' has been "copied" to '`p1`' and '`p2`'.

```
int main(void)
{
   int x, y, *p1, *p2;
   p1 = &x;
   /* p1 ↦ {x} */
   foo(p1);
   p2 = &y;
   /* p2 ↦ {y} */
}
```

Renaming of variables can clearly be done automatically. **End of Example**

The transformation fails on indirect initializations, *e.g.* an assignment '`*q = &x;`', where '`q`' points to a pointer variable.[6]

4.2.3 Intra- and inter-procedural analysis

Intra-procedural analysis is concerned with the data flow in function bodies, and makes worst-call assumption about function calls. In this chapter we shall use 'intra-procedural' in a more strict meaning: functions are analysed context-independently. *Inter-procedural*

[5]As opposed to Fortran, that tends to use "long" functions.

[6]All flow-sensitive analyses will gain from this transformation, including binding-time analysis.

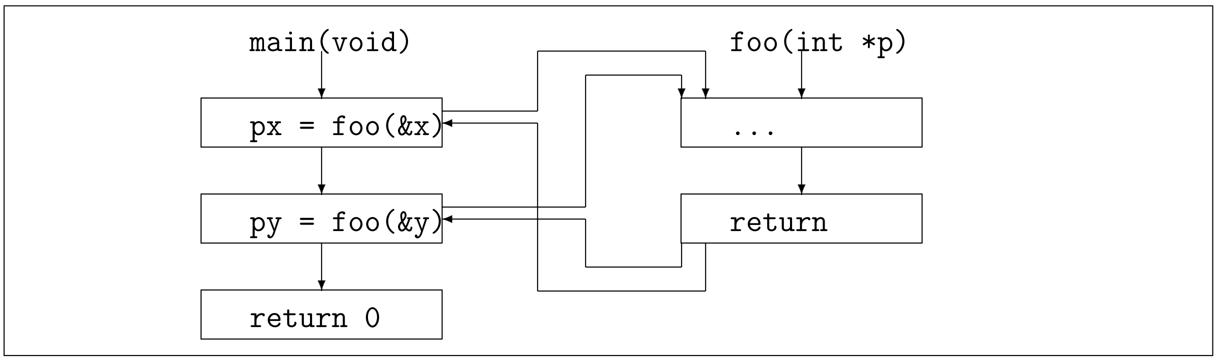

Figure 34: Inter-procedural call graph for program in Example 4.6

analysis infers information under consideration of call contexts. Intra-procedural analysis is also called *monovariant* or *sticky*, and inter-procedural analysis is also known as *polyvariant*.

Example 4.6 Consider the following program.

```
int main(void)              int *foo(int *p)
{                           {
   int x,y,*px,*py;            ...
   px = foo(&x);               return p;
   py = foo(&y);            }
   return 0;
}
```

An intra-procedural analysis merges the contexts of the two calls and computes the point-to information $[px, py \mapsto \{x, y\}]$. An inter-procedural analysis differentiates between to two calls. Figure 34 illustrates the inter-procedural call graph. **End of Example**

Inter-procedural analysis improves the precision of intra-procedural analysis by preventing calls to interfere. Consider Figure 34 that depicts the inter-procedural call graphs of the program in Example 4.6. The goal is that the value returned by the first call is not erroneous propagated to the second call, and vice versa. Information must only be propagated through *valid* or *realizable* program paths [Sharir and Pnueli 1981]. A control-path is realizable when the inter-procedural exit-path corresponds to the entry path.

4.2.4 Use of inter-procedural information

Inter-procedural analysis is mainly concerned with the *propagation* of value-flow information through functions. Another aspect is the *use* of the inferred information, *e.g.* for optimization, or to drive other analyses. Classical inter-procedural analyses produce a summary for each function, that is, all calls are merged. Clearly, this degrades the number of possible optimizations.

Example 4.7 Suppose we apply inter-procedural constant propagation to a program containing the calls '`bar(0)`' and '`bar(1)`'. Classical analysis will merge the two calls and henceforth classify the parameter for 'non-const', ruling out *e.g.* compile-time execution of an `if` statement [Callahan *et al.* 1986]. **End of Example**

An aggressive approach would be either to *inline* functions into the caller or to *copy* functions according to their use. The latter is also known as *procedure cloning* [Cooper *et al.* 1993,Hall 1991].

We develop a flexible approach where each function is annotated with both context specific information and a summary. At a later stage the function can then be cloned, if so desired. We return to this issue in Chapter 6, and postpone the decision whether to clone a function or not.[7]

We will assume that a program's static-call graph is available. Recall that the static-call graph approximates the invocation of functions, and assigns a *variant* number to functions according to the call contexts. For example, if a function is called in n contexts, the function has n variants. Even though function not are textually copied according to contexts, it is useful to imagine that n variants of the function's parameters and local variables exist. We denote by v^i the variable corresponding to the i'th variant.

4.2.5 May or must?

The *certainty* of a pointer abstraction can be characterized by *may* or *must.* A *may point-to* analysis computes for every pointer set of abstract locations that the pointer *may* point to at runtime. A *must point-to* analysis computes for every pointer a set of abstract locations that the pointer *must* point to.

May and must analysis is also known as *existential* and *universal* analysis. In the former case, there must exists a path where the point-to relation is valid, in the latter case the point-to relation must be valid on all paths.

Example 4.8 Consider live-variable analysis of the expression '`x= *p`'. Given must point-to information $[p \mapsto \{y\}]$, '`y`' can be marked "live". On the basis of may point to information $[p \mapsto \{y, z\}]$, both '`y` and '`z`' must be marked "live". **End of Example**

We shall only consider may point-to analysis in this chapter.

4.3 Pointer analysis of C

In this section we briefly consider pointer analysis of some of the more intricate features of C such separate compilation, external variables and non-strictly complying expressions, *e.g.* type casts, and their interaction with pointer analysis.

4.3.1 Structures and unions

C supports user-defined structures and unions. Recall from Section 2.3.3 that struct variables sharing a common type definition are separated (are given different names) during parsing. After parsing, a value-flow analysis unions (the types of) objects that (may) flow together.

[7]Relevant information includes number of calls, the size of the function, number of calls in the functions.

Example 4.9 Given definitions '`struct S { int *p;} s,t,u;`', variants of the struct type will be assigned to the variables, *e.g.* '`s`' will be assigned the type '`struct S1`'. Suppose that the program contains the assignment '`t = s`'. The value-flow analysis will then merge the type definitions such that '`s`' and '`t`' are given the same type ('`struct S1`', say), whereas '`u`' is given the type '`struct S3`', say. **End of Example**

Observe: struct variables of different type *cannot flow together*. Struct variables of the same type *may flow together*. We exploit this fact the following way.

Point-to information for field members of a struct variable is associated with the *definition* of a struct; not the struct objects. For example, the point to information for member '`s.p`' (assuming the definitions from the Example above) is represented by '`S1.p`', where '`S1`' is the "definition" of '`struct S1`'. The definition is common for all objects of that type. An important consequence: in the case of an assignment '`t = s`', the fields of '`t`' do not need to be updated with respect to '`s`' — the value-flow analysis have taken care of this.

Hence, the pointer analysis is factorized into the two sub-analyses

1. a (struct) value-flow analysis, and
2. a point-to propagation analysis

where this chapter describes the propagation analysis. We will (continue to) use the term pointer analysis for the propagation analysis.

Recall from Chapter 2 that some initial members of unions are truly shared. This is of importance for pointer analysis if the member is of pointer type. For simplicity we we will not take this aspect into account. The extension is straightforward, but tedious to describe.

4.3.2 Implementation-defined features

A C program can comply to the Standard in two ways. A *strictly conforming* program shall not depend on implementation-defined behavior but a *conforming* program is allowed to do so. In this section we consider type casts that (in most cases) are non-strictly conforming.

Example 4.10 Cast of an integral value to a pointer or conversely is an implementation-defined behaviour. Cast of a pointer to a pointer with less alignment requirement and back again, is strictly conforming [ISO 1990]. **End of Example**

Implementation-defined features cannot be described accurately by an architecture-independent analysis. We will approximate pointers that may point to any object by the unique abstract location 'Unknown'.

Definition 4.1 *Let '*`p`*' be a pointer. If a pointer abstraction maps '*`p`*' to* Unknown*,* $[p \mapsto \text{Unknown}]$*, when '*`p`*' may point to* all *accessible objects at runtime.* □

The abstract location 'Unknown' corresponds to "know nothing", or "worst-case".

Example 4.11 The goal parameters of a program must be described by 'Unknown', *e.g.* the '`main`' function

```
int main(int argc, char **argv)
{ ... }
```

is approximated by [`argv` $\mapsto$ {Unknown}]. **End of Example**

In this chapter we will *not* consider the `setjmp`' and '`longjmp`' macros.

4.3.3 Dereferencing unknown pointers

Suppose that a program contains an assignment through an Unknown pointer, *e.g.* '`*p = 2`', where [$p \mapsto$ {Unknown}]. In the case of live-variable analysis, this implies that worst-case assumptions must be made. However, the problem also affects the pointer analysis.

Consider an assignment '`*q = &x`', where '`q`' is unknown. This implies after the assignment, *all* pointers may point to '`x`'. Even worse, an assignment '`*q = p`' where '`p`' is unknown renders *all* pointers unknown.

We shall proceed as follows. If the analysis reveals that an Unknown pointer may be dereferenced in the left hand side of an assignment, the analysis stops with "worst-case" message. This corresponds to the most inaccurate pointer approximation possible. Analyses depending on pointer information must make worst-case assumptions about the pointer usage.

For now we will assume that Unknown pointers are *not* dereferenced in the left hand side of an assignment. Section 4.8 describes handling of the worst-case behaviour.

4.3.4 Separate translation units

A C program usually consists of a collection of translation units that are compiled separately and linked to an executable. Each file may refer to variables defined in other units by the means of '`extern`' declarations. Suppose that a pointer analysis is applied to a single module.

This has two consequences. Potentially, global variables may be modified by assignments in other modules. To be safe, worst-case assumptions, *i.e.* Unknown, about global variables must be made. Secondly, functions may be called from other modules with unknown parameters. Thus, to be safe, all functions must be approximated by Unknown.

To obtain results other than trivial we shall avoid separate analysis, and assume that "relevant" translation units are merged; *i.e.* we consider solely monolithic programs. The subject of Chapter 7 is separate program analysis, and it outlines a separate pointer analysis based on the development in this chapter.

Constraint 4.1 *i*) No global variables of pointer type may be modified by other units. *ii*) Functions are assumed to be static to the translation unit being analyzed.

It is, however, convenient to sustain the notion of an object being "external". For example, we will describe the function '`strdup()`' as returning a pointer to an 'Unknown' object.

4.4 Safe pointer abstractions

A *pointer abstraction* is a map from abstract program objects (variables) to sets of abstract locations. An abstraction is *safe* if for every object of pointer type, the set of concrete addresses it may contain at runtime is safely described by the set of abstract locations. For example, if a pointer '`p`' may contain the locations l_x (location of '`x`') and l_g (location of '`g`') at runtime, a safe abstraction is $p \mapsto \{x, g\}$.

In this section we define *abstract locations* and make precise the notion of *safety*. We present a specification that can be employed to check the safety of an abstraction. The specification serves as foundation for the development of a constraint-based pointer analysis.

4.4.1 Abstract locations

A pointer is a variable containing the distinguished constant '`NULL`' or an address. Due to casts, a pointer can (in principle) point to an arbitrary address. An *object* is a set of logically related locations, *e.g.* four bytes representing an integer value, or n bytes representing a struct value. Since pointers may point to functions, we will also consider functions as objects.

An object can either be allocated on the program stack (local variables), at a fixed location (strings and global variables), in the code space (functions), or on the heap (runtime allocated objects). We shall only be concerned with the run time allocated objects brought into existence via '`alloc()`' calls. Assume that all calls are labeled uniquely.[8] The label l of an '`alloc`^{l}`()`' is used to denote the set of (anonymous) objects allocated by the '`alloc`^{l}`()`' call-site. The label l may be thought of as a pointer of a relevant type.

Example 4.12 Consider the program lines below.

```
int x, y, *p, **q, (*fp)(void);
struct S *ps;
p = &x;
q = &p;
*q = &y;
fp = &foo;
ps = alloc¹(S);
```

We have: $[p \mapsto \{x, y\}, q \mapsto \{p\}, fp \mapsto \{foo\}, ps \mapsto \{1\}]$. **End of Example**

Consider an application of the address operator &. Similar to an '`alloc()`' call, it "returns" a pointer to an object. To denote the set of objects the application "returns", we assume assume a unique labeling. Thus, in '`p = &`^{2}`x`' we have that '`p`' points to the same object as the "pointer" '2', that is, x.

Definition 4.2 *The set of* abstract locations ALoc *is defined inductively as follows:*

[8]Objects allocated by the means of 'malloc' are considered 'Unknown'.

- *If v is the name of a global variable: $v \in \mathrm{ALoc}$.*
- *If v is a parameter of some function with n variants: $v^i \in \mathrm{ALoc}, i = 1, \ldots, n$.*
- *If v is a local variable in some function with n variants: $v^i \in \mathrm{ALoc}, i = 1, \ldots, n$.*
- *If s is a string constant: $s \in \mathrm{ALoc}$.*
- *If f is the name of a function with n variants: $f^i \in \mathrm{ALoc}, i = 1, \ldots, n$.*
- *If f is the name of a function with n variants: $f_0^i \in \mathrm{ALoc}, i = 1, \ldots, n$.*
- *If l is the label of an alloc in a function with n variants: $l^i \in \mathrm{ALoc}, i = 1, \ldots, n$.*
- *If l is the label of an address operator in a function with n variants: $l^i \in \mathrm{ALoc}$.*
- *If $o \in \mathrm{ALoc}$ denotes an object of type "array": $o[] \in \mathrm{ALoc}$.*
- *If S is the type name of a struct or union type: $S \in \mathrm{ALoc}$.*
- *If $S \in \mathrm{ALoc}$ is of type "struct" or "union": $S.i \in \mathrm{ALoc}$ for all fields i of S.*
- $\mathrm{Unknown} \in \mathrm{ALoc}$.

Names are assumed to be unique. □

Clearly, the set ALoc is finite for all programs. The analysis maps a pointer into an element of the set $\wp(\mathrm{ALoc})$. The element Unknown denotes an arbitrary (unknown) address. This means that the analysis abstracts as follows.

Function invocations are collapsed according to the program's static-call graph (see Chapter 2). This means for a function f with n variants, only n instances of parameters and local variables are taken into account. For instance, due to the 1-limit imposed on recursive functions, all instances of a parameters in a recursive function invocation chain are identified. The location f_0 associated with function f denotes an *abstract return location*, *i.e.* a unique location where f "delivers" its result value.

Arrays are treated as aggregates, that is, all entries are merged. Fields of struct objects of the same name are merged, *e.g.* given definition '`struct S { int x;} s,t`', fields '`s.x`' and '`t.x`' are collapsed.

Example 4.13 The merging of struct fields may seen excessively conservatively. However, recall that we assume programs are type-separated during parsing, and that a value-flow analysis is applied that identifier the type of struct objects that (may) flow together, see Section 2.3.3. **End of Example**

The unique abstract location Unknown denotes an arbitrary, unknown address, which both be valid or illegal.

Even though the definition of abstract locations actually is with respect to a particular program, we will continue to use ALoc independently of programs. Furthermore, we will assume that the type of the object, an abstract location denotes, is available. For example, we write "if $S \in \mathrm{ALoc}$ is of struct type", for "if the object $S \in \mathrm{ALoc}$ denotes is of struct type". Finally, we implicitly assume a binding from a function designator to the parameters. If f is a function identifier, we write $f\!:\!x_i$ for the parameter x_i of f.

4.4.2 Pointer abstraction

A *pointer abstraction* $\tilde{\mathcal{S}} : \mathrm{ALoc} \to \wp(\mathrm{ALoc})$ is a map from abstract locations to sets of abstract locations.

Example 4.14 Consider the following assignments.

```
int *p, *q;
extern int *ep;
p = (int *)0xabcd;
q = (int *)malloc(100*sizeof(int));
r = ep;
```

The pointer '`p`' is assigned a value via a non-portable cast. We will approximate this by Unknown. Pointer '`q`' is assigned the result of '`malloc()`'. In general, pointers returned by external functions are approximated by Unknown. Finally, the pointer '`r`' is assigned the value of an external variable. This is also approximated by Unknown.

A refinement would be to approximate the content of external pointers by a unique value Extern. Since we have no use for this, besides giving more accurate warning messages, we will not pursue this. **End of Example**

A pointer abstraction $\tilde{\mathcal{S}}$ must fulfill the following requirements which we justify below.

Definition 4.3 *A* pointer abstraction $\tilde{\mathcal{S}} : \mathrm{ALoc} \to \wp(\mathrm{ALoc})$ *is a map satisfying:*

1. *If* $o \in \mathrm{ALoc}$ *is of base type:* $\tilde{\mathcal{S}}(o) = \{\mathrm{Unknown}\}$.
2. *If* $s \in \mathrm{ALoc}$ *is of struct/union type:* $\tilde{\mathcal{S}}(s) = \{\}$.
3. *If* $f \in \mathrm{ALoc}$ *is a function designator:* $\tilde{\mathcal{S}}(f) = \{\}$.
4. *If* $a \in \mathrm{ALoc}$ *is of type array:* $\tilde{\mathcal{S}}(a) = \{a[]\}$.
5. $\tilde{\mathcal{S}}(\mathrm{Unknown}) = \mathrm{Unknown}$.

□

The first condition requires that objects of base types are abstracted by Unknown. The motivation is that the value may be cast into a pointer, and is hence Unknown (in general). The second condition stipulates that the abstract value of a struct object is the empty set. Notice that a struct object is uniquely identified its type. The fourth condition requires that an array variable points to the content.[9] Finally, the content of an unknown location is unknown.

Define for $s \in \mathrm{ALoc} \setminus \{\mathrm{Unknown}\} : \{s\} \subseteq \{\mathrm{Unknown}\}$. Then two pointer abstractions are ordered by set inclusion. A program has a minimal pointer abstraction. Given a program, we desire a minimal *safe* pointer abstraction.

[9]In reality, 'a' in 'a[10]' is not an lvalue. It is, however, convenient to consider 'a' to be a pointer to the content.

4.4.3 Safe pointer abstraction

Intuitively, a pointer abstraction for a program is *safe* if for all input, every object a pointer may point to at runtime is captured by the abstraction.

Let the abstraction function $\alpha : \mathrm{Loc} \to \mathrm{ALoc}$ be defined the obvious way. For example, if l_x is the location of parameter '`x`' in an invocation of a function '`f`' corresponding to the i'th variant, then $\alpha(l_x) = x^i$. An execution path from the initial program point p_0 and an initial program store $\mathcal{S}_0$ is denoted by

$$\langle p_0, \mathcal{S}_0 \rangle \to \cdots \to \langle p_n, \mathcal{S}_n \rangle$$

where $\mathcal{S}_n$ is the store at program point p_n.

Let p be a program and $\mathcal{S}_0$ an initial store (mapping the program input to the parameters of the goal function). Let p_n be a program point, and $\mathcal{L}_n$ the locations of all visible variables. A *pointer abstraction* $\tilde{\mathcal{S}}$ is *safe* with respect to p if

$$l \in \mathcal{L}_n : \alpha(\mathcal{S}_n(l)) \subseteq \tilde{\mathcal{S}}(\alpha(l))$$

whenever $\langle p_0, \mathcal{S}_0 \rangle \to \cdots \to \langle p_n, \mathcal{S}_n \rangle$.

Every program has a *safe* pointer abstraction. Define $\tilde{\mathcal{S}}_{triv}$ such that it fulfills Definition 4.3, and extend it such that for all $o \in \mathrm{ALoc}$ where o is of pointer type, $\tilde{\mathcal{S}}_{triv}(o) = \{\mathrm{Unknown}\}$. Obviously, it is a safe — and useless — abstraction.

The definition of safety above considers only monolithic programs where no external functions nor variables exist. We are, however, interested in analysis of translation units where parts of the program may be undefined.

Example 4.15 Consider the following piece of code.

```
extern int *o;
int *p, **q;
q = &o;
p = *q;
```

Even though '`o`' is an external variable, it can obviously be established that $[q \mapsto \{o\}]$. However, '`p`' must inevitably be approximated by $[p \mapsto \{\mathrm{Unknown}\}]$. **End of Example**

Definition 4.4 *Let $p \equiv m_1, \ldots, m_m$ be a program consisting of the modules m_i. A pointer abstraction $\tilde{\mathcal{S}}$ is* safe *for m_{i_0} if for all program points p_n and initial stores $\mathcal{S}_0$ where $\langle p_0, \mathcal{S} \rangle \to \cdots \to \langle p_n, \mathcal{S}_n \rangle$, then:*

- *for $l \in \mathcal{L}_n$: $\alpha(\mathcal{S}_n(l)) \subseteq \tilde{\mathcal{S}}(\alpha(l))$ if l is defined in m_{i_0},*
- *for $l \in \mathcal{L}_n$: $\tilde{\mathcal{S}}(l) = \{\mathrm{Unknown}\}$ if l is defined in $m_i \neq m_{i_0}$*

where $\mathcal{L}_n$ is the set of visible variables at program point n. □

For simplicity we regard $a[]$, given an array a, to be a "visible variable", and we regard the labels of '`alloc()`' calls to be "pointer variables.

Example 4.16 Suppose that we introduced an abstract location Extern to denote the contents of external variables. Example 4.15 would then be abstracted by: $[p \mapsto \text{Extern}]$. There is no operational difference between Extern and Unknown. **End of Example**

We will compute an approximation to a safe pointer abstraction. For example, we abstract the result of an implementation-defined cast, *e.g.* '`(int *)x`' where '`x`' is an integer variable, by Unknown, whereas the definition may allow a more accurate abstraction.

4.4.4 Pointer analysis specification

We specify a *flow-insensitive* (summary), *intra-procedural* pointer analysis. We postpone extension to inter-procedural analysis to Section 4.6.

The specification can be employed to check that a given pointer abstraction $\tilde{\mathcal{S}}$ is *safe* for a program. Due to lack of space we only present the rules for declarations and expressions (the interesting cases) and describe the other cases informally. The specification is in the form of inference rules $\tilde{\mathcal{S}} \vdash p : \bullet$.

We argue (modulo the omitted part of the specification) that if the program fulfills the rules in the context of a pointer abstraction $\tilde{\mathcal{S}}$, then $\tilde{\mathcal{S}}$ is a safe pointer abstraction. Actually, the rules will also fail if $\tilde{\mathcal{S}}$ is not a pointer abstraction, *i.e.* does not satisfy Definition 4.3. Let $\tilde{\mathcal{S}}$ be given.

Suppose that $d \equiv x : T$ is a definition (*i.e.*, not an 'extern' declaration). The safety of $\tilde{\mathcal{S}}$ with respect to d depends on the type T.

Lemma 4.1 *Let* $d \in \text{Decl}$ *be a definition. Then* $\tilde{\mathcal{S}} : \text{ALoc} \rightarrow \wp(\text{ALoc})$ *is a* pointer abstraction *with respect to* d *if*

$$\tilde{\mathcal{S}} \vdash^{pdecl} d : \bullet$$

where $\vdash^{pdecl}$ *is defined in Figure 35,* and $\tilde{\mathcal{S}}(\text{Unknown}) = \{\text{Unknown}\}$.

Proof It is straightforward to verify that Definition 4.3 is fulfilled. □

To the right of Figure 35 the rules for external variables are shown. Let $d \equiv x : T$ be an (extern) declaration. Then $\tilde{\mathcal{S}}$ is a *pointer abstraction* for d if $\tilde{\mathcal{S}} \vdash^{petype} \langle T, l\rangle : \bullet$. Notice the rules require external pointers to be approximated by Unknown, as stipulated by Definition 4.4.

The (omitted) rule for function definitions $T_f\ f(d_i)\{d_j\ S_k\}$ (would) require $\tilde{\mathcal{S}}(f) = \{f_0\}$.

Since we specify a flow-insensitive analysis, the safety of a pointer abstraction with respect to an expression e is independent of program points. A map $\tilde{\mathcal{S}} : \text{ALoc} \rightarrow \wp(\text{ALoc})$ is a *pointer abstraction* with respect to an expression e, if it is a pointer abstraction with respect to the variables occurring in e.

Lemma 4.2 *Let* $e \in \text{Expr}$ *be an expression and* $\tilde{\mathcal{S}}$ *a pointer abstraction with respect to* e. *Then* $\tilde{\mathcal{S}}$ *is* safe *provided there exist* $V \in \wp(\text{ALoc})$ *such*

[decl]	$\dfrac{\vdash^{ctype} \langle T, x\rangle : \bullet}{\vdash^{ptype} d : \bullet} \quad d \equiv x : T$	$\dfrac{\vdash^{petype} \langle T, x\rangle : \bullet}{\vdash^{pdecl} \texttt{extern}\ x : T : \bullet} \quad d \equiv x : T$
[base]	$\dfrac{\tilde{\mathcal{S}}(l) = \{\text{Unknown}\}}{\tilde{\mathcal{S}} \vdash^{ptype} \langle\langle \tau_b\rangle, l\rangle : \bullet}$	$\dfrac{\tilde{\mathcal{S}}(l) = \{\text{Unknown}\}}{\tilde{\mathcal{S}} \vdash^{petype} \langle\langle \tau_b\rangle, l\rangle : \bullet}$
[struct]	$\dfrac{\tilde{\mathcal{S}}(l) = \{\}}{\tilde{\mathcal{S}} \vdash^{ptype} \langle\langle \texttt{struct S}\rangle, l\rangle : \bullet}$	$\dfrac{\tilde{\mathcal{S}}(l) = \{\text{Unknown}\}}{\tilde{\mathcal{S}} \vdash^{petype} \langle\langle \texttt{struct S}\rangle, l\rangle : \bullet}$
[union]	$\dfrac{\tilde{\mathcal{S}}(l) = \{\}}{\tilde{\mathcal{S}} \vdash^{ptype} \langle\langle \texttt{union U}\rangle, l\rangle : \bullet}$	$\dfrac{\tilde{\mathcal{S}}(l) = \{\text{Unknown}\}}{\tilde{\mathcal{S}} \vdash^{petype} \langle\langle \texttt{union U}\rangle, l\rangle : \bullet}$
[ptr]	$\vdash^{ptype} \langle\langle * \rangle\, T, l\rangle : \bullet$	$\dfrac{\tilde{\mathcal{S}}(l) = \{\text{Unknown}\}}{\tilde{\mathcal{S}} \vdash^{ptype} \langle\langle * \rangle\, T, l\rangle : \bullet}$
[array]	$\dfrac{\vdash^{ptype} \langle T, l[]\rangle : \bullet \quad \tilde{\mathcal{S}}(l) = \{l[]\}}{\vdash^{ptype} \langle\langle [n]\rangle\, T, l\rangle : \bullet}$	$\dfrac{\vdash^{petype} \langle T, l[]\rangle : \bullet \quad \tilde{\mathcal{S}}(l) = \{l[]\}}{\vdash^{petype} \langle\langle [n]\rangle\, T, l\rangle : \bullet}$
[fun]	$\vdash^{ptype} \langle\langle (d_i) T\rangle, l\rangle : \bullet$	$\vdash^{ptype} \langle\langle (d_i) T\rangle, l\rangle : \bullet$

Figure 35: Pointer abstraction for declarations

$$\tilde{\mathcal{S}} \vdash^{pexp} e : V$$

where $\vdash^{pexp}$ *is defined in Figure 36.*

Intuitively, the the rules infer the *lvalues* of the expression e. For example, the lvalue of a variable v is $\{v\}$; recall that we consider intra-procedural analysis only.[10]

An informal justification of the lemma is given below. We omit a formal proof.

Justification A formal proof would be by induction after "evaluation length". We argue that if $\tilde{\mathcal{S}}$ is safe before evaluation of e, it is also safe after.

A constant has an Unknown lvalue, and the lvalue of a string is given by its name. The motivation for approximating the lvalue of a constant by Unknown, rather than the empty set, if obvious from the following example: '`p = (int *)12`'. The lvalue of a variable is approximated by its name.

Consider a struct indexing $e.i$. Given the type S of the objects the subexpression denotes, the lvalues of the fields are $S.i$. The rules for pointer dereference and array indexing use the pointer abstraction to describe the lvalue of the dereferenced objects. Notice: if '`p`' points to '`x`', that is $\tilde{\mathcal{S}}(p) = \{x\}$, when the lvalue of '`*p`' is the lvalue of '`x`' which is approximated by $\{x\}$. The rule for the address operator uses the label as a "placeholder" for the indirection created.

The effect of unary and binary operator applications is described by the means of $\tilde{O} : \text{Op} \times \wp(\text{ALoc})^* \to \wp(\text{ALoc})$. We omit a formal specification.

Example 4.17 Suppose that '`p`' and '`q`' both point to an array and consider pointer subtraction '`p - q`'.[11] We have $\tilde{O}(-_{*int,*int}, \{p\}, \{q\}) = \{\text{Unknown}\}$ since the result is an integer. Consider now '`p - 1`'. We then get $\tilde{O}(-_{*int,int}, \{p\}, \{\text{Unknown}\}) = \{p\}$ since pointer arithmetic is not allowed to shuffle a pointer outside an array. **End of Example**

[10]That is, there is one "variant" of each function.

[11]Recall that operator overloading is assumed resolved during parsing.

An external function delivers its result in an unknown location (and the result itself is unknown).

Consider the rules for functions calls. The content of the argument's abstract lvalue must be contained in the description of the formal parameters.[12] The result of the application is returned in the called function's abstract return location. In the case of indirect calls, all possible functions are taken into account.

Example 4.18 In case of the program fragment

```
int (*fp)(int), x;
fp = &foo;
fp = &bar;
(*fp)(&x)
```

where '`foo()`' and '`bar()`' are two functions taking an integer pointer as a parameter, we have:

$$[\mathit{fp} \mapsto \{\mathit{foo}, \mathit{bar}\}]$$

due to the first two applications of the address operator, and

$$[\mathit{foo{:}x} \mapsto \{x\}, \mathit{bar{:}x} \mapsto \{x\}]$$

due to the indirect call. The 'lvalue' of the call is $\{\mathit{foo}_0, \mathit{bar}_0\}$. **End of Example**

The rules for pre- and post increment expressions are trivial.

Consider the rule for assignments. The content of locations the left hand side must contain the content of the right hand side expression. Recall that we assume that no Unknown pointers are dereferenced.

Example 4.19 Consider the following assignments

```
extern int **q;
int *p;
*q = p;
```

Since '`q`' is extern, it is Unknown what it points to. Thus, the assignment may assign the pointer '`p`' to an Unknown object (of pointer type). This extension is shown in Section 4.3.3. **End of Example**

The abstract lvalue of a comma expression is determined by the second subexpression. A `sizeof` expression has no lvalue and is approximated by Unknown.

Finally, consider the rule for casts. It uses the function Cast : Type $\times$ Type $\times$ $\wp(\text{ALoc}) \rightarrow \wp(\text{ALoc})$ defined as follows.

[12]Recall that we consider intra-procedural, or sticky analysis.

[const] $\tilde{\mathcal{S}} \vdash^{pexp} c : \{\text{Unknown}\}$

[string] $\tilde{\mathcal{S}} \vdash^{pexp} s : \{s\}$

[var] $\tilde{\mathcal{S}} \vdash^{pexp} v : \{v\}$

[struct] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1 \quad \text{TypOf}(o \in O_1) = \langle \text{struct S} \rangle}{\tilde{\mathcal{S}} \vdash^{pexp} e_1.i : \{S.i\}}$

[indr] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1}{\tilde{\mathcal{S}} \vdash^{pexp} \texttt{*}e_1 : \bigcup_{o \in O_1} \tilde{\mathcal{S}}(o)}$

[array] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1 \qquad \tilde{\mathcal{S}} \vdash^{pexpr} e_2 : O_2}{\tilde{\mathcal{S}} \vdash^{pexp} e_1\texttt{[}e_2\texttt{]} : \bigcup_{o \in O_1} \tilde{\mathcal{S}}(o)}$

[address] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1 \qquad \tilde{\mathcal{S}}(l) \supseteq O_1}{\tilde{\mathcal{S}} \vdash^{pexp} \texttt{\&}^l e_1 : \{l\}}$

[unary] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1}{\tilde{\mathcal{S}} \vdash^{pexp} o\ e_1 : \tilde{O}(o, O_1)}$

[binary] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_i : O_i}{\tilde{\mathcal{S}} \vdash^{pexp} e_1\ op\ e_2 : \tilde{O}(o, O_i)}$

[alloc] $\tilde{\mathcal{S}} \vdash^{pexp} \texttt{alloc}^l(T) : \{l\}$

[extern] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_i : O_i}{\tilde{\mathcal{S}} \vdash^{pexp} ef\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} : \{\text{Unknown}\}}$

[user] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_i : O_i \qquad \tilde{\mathcal{S}}(f{:}x_i) \supseteq \tilde{\mathcal{S}}(O_i)}{\tilde{\mathcal{S}} \vdash^{pexp} f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} : \tilde{\mathcal{S}}(f_0)}$

[call] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_0 : O_0 \qquad \forall o \in O_0 : \tilde{\mathcal{S}}(o{:}x_i) \supseteq \tilde{\mathcal{S}}(O_i)}{\tilde{\mathcal{S}} \vdash^{pexp} e_0(e_1, \ldots, e_n) : \bigcup_{o \in O_0} \tilde{\mathcal{S}}(o_0)}$

[preinc] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1}{\tilde{\mathcal{S}} \vdash^{pexp} \texttt{++}e_1 : O_1}$

[postinc] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1}{\tilde{\mathcal{S}} \vdash^{pexp} e_1\texttt{++} : O_1}$

[assign] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1 \qquad \tilde{\mathcal{S}} \vdash^{pexp} e_2 : O_2 \quad \forall o \in O_1 : \tilde{\mathcal{S}}(o) \supseteq \tilde{\mathcal{S}}(O_2)}{\tilde{\mathcal{S}} \vdash^{pexp} e_1\ aop\ e_2 : O_2}$

[comma] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1 \qquad \tilde{\mathcal{S}} \vdash^{pexp} e_2 : O_2}{\tilde{\mathcal{S}} \vdash^{pexp} e_1, e_2 : O_2}$

[sizeof] $\tilde{\mathcal{S}} \vdash^{pexp} \texttt{sizeof}(T) : \{\text{Unknown}\}$

[cast] $\dfrac{\tilde{\mathcal{S}} \vdash^{pexp} e_1 : O_1}{\tilde{\mathcal{S}} \vdash^{pexp} (T)e_1 : \text{Cast}(T, \text{TypOf}(e_1), O_1)}$

Figure 36: Pointer abstraction for expressions

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{Cast}(T_{to}, T_{from}, O_{from}) = \text{case } (T_{to}, T_{from}) \text{ of}} \\
\quad (\langle\tau_b\rangle, \langle\tau_b'\rangle) & : O_{from} & \\
\quad (\langle*\rangle\, T, \langle\tau_b\rangle) & : \{\text{Unknown}\} & \\
\quad (\langle\tau_b\rangle, \langle*\rangle\, T) & : \{\text{Unknown}\} & \\
\quad (\langle*\rangle\, T, \langle*\rangle\, \langle\texttt{struct S}\rangle) & : \begin{cases} \{o.1 \mid o \in O_{from}\} & T \text{ type of first member of } S \\ O_{from} & \text{Otherwise} \end{cases} & \\
\quad (\langle*\rangle\, T', \langle*\rangle\, T'') & : O_{from} &
\end{array}$$

Casts between base types do not change an object's lvalue. Casts from a pointer type to an integral type, or the opposite, is implementation-defined, and approximated by Unknown.

Recall that a pointer to a struct object points, when suitably converted, also to the first member. This is implemented by the case for cast from struct pointer to pointer. We denote the name of the first member of S by '1'. Other conversions do not change the lvalue of the referenced objects. This definition is in accordance with the Standard [ISO 1990, Paragraph 6.3.4]. **End of Justification**

The specification of statements uses the rules for expressions. Further, in the case of a '`return` e':

$$\frac{\tilde{\mathcal{S}} \vdash^{pexp} e : O \quad \tilde{\mathcal{S}}(f_0) \supseteq \tilde{\mathcal{S}}(O)}{\tilde{\mathcal{S}} \vdash^{pstmt} \texttt{return } e : \bullet}$$

which specifies that the abstract return location of function f (encapsulating the statement) must containt the value of the expression e.

We conjecture that given a program p and a map $\tilde{\mathcal{S}} : \mathrm{ALoc} \to \wp(\mathrm{ALoc})$, then $\tilde{\mathcal{S}}$ is a *safe pointer abstraction* for p iff the rules are fulfilled.

4.5 Intra-procedural pointer analysis

This section presents a *constraint-based* formulation of the pointer analysis specification. The next section extends the analysis to an *inter-procedural* analysis, and Section 4.7 describes constraint solving.

4.5.1 Pointer types and constraint systems

A *constraint system* is defined as a set of constraints over *pointer types*. A solution to a constraint system is a substitution from pointer type variables to sets of abstract locations, such that all constraints are satisfied.

The syntax of a *pointer type* T is defined inductively by the grammar

$$\begin{array}{rcll}
\mathcal{T} & ::= & \{o_j\} & \text{locations} \\
 & | & *\mathcal{T} & \text{deference} \\
 & | & \mathcal{T}.i & \text{indexing} \\
 & | & (\mathcal{T}) \to \mathcal{T} & \text{function} \\
 & | & T & \text{type variable}
\end{array}$$

where $o_j \in$ ALoc and i is an identifier. A pointer type can be a *set* of abstract locations, a *dereference type*, an *indexing* type, a *function* type, or a *type variable.* Pointer types $\{o_j\}$ are *ground types.* We use $\mathcal{T}$ to range over pointer types.

To every object $o \in$ ALoc of non-functional type we assign a type variable T_o; this includes the abstract return location f_0 for a function f. To every object $f \in$ ALoc of function type we associate the type $(T_d) \to T_{f_0}$, where T_d are the type variables assigned to parameters of f. To every type specifier τ we assign a type variable T_τ.

The aim of the analysis is to instantiate the type variables with an element from $\wp(\text{ALoc})$, such that the map $[o \mapsto T_o]$ becomes a safe pointer abstraction.

A *variable assignment* is a substitution S : TVar $\to$ PType from type variables to ground pointer types. Application of a substitution S to a type $\mathcal{T}$ is denoted by juxtaposition $S \cdot \mathcal{T}$. The *meaning* of a pointer type is defined relatively to a variable assignment.

Definition 4.5 *Suppose that S is a variable assignment. The* meaning *of a pointer type $\mathcal{T}$ is defined by*

$$\begin{array}{lcl}
[\![O]\!]\ S & = & O \\
[\![*\mathcal{T}]\!]\ S & = & \bigcup_o ST_o, \quad o \in [\![\mathcal{T}]\!]\ S \\
[\![\mathcal{T}.i]\!]\ S & = & \bigcup_o \{S(U.i) \mid \text{TypOf}(o) = \langle \text{struct U} \rangle\} \quad o \in [\![\mathcal{T}]\!]\ S \\
[\![(\mathcal{T}_i) \to \mathcal{T}]\!]\ S & = & ([\![\mathcal{T}_i]\!]\ S) \to [\![\mathcal{T}]\!]\ S \\
[\![T]\!]\ S & = & ST
\end{array}$$

where T_o is the unique type variable associated with object o. □

The meaning of a deference type $*\mathcal{T}$ is determined by the variable assignment. Intuitively, if $\mathcal{T}$ denotes objects $\{o_i\}$, the meaning is the contents of those objects: ST_{o_i}. In the case of an indexing $\mathcal{T}.i$, the meaning equals content of the fields of the object(s) $\mathcal{T}$ denote.

A *constraint system* is a multi-set of formal inclusion constraints

$$\mathcal{T} \supseteq \mathcal{T}$$

over pointer types $\mathcal{T}$. We use $\mathcal{C}$ to denote constraint systems.

A *solution* to a constraint system $\mathcal{C}$ is a *substitution* S : TVar $\to$ PType from type variables to ground pointer types which is the identity on variables but those occurring in $\mathcal{C}$, such that all constraints are satisfied.

Definition 4.6 *Define the relation $\supseteq^*$ by $O_1 \supseteq^* O_2$ iff $O_1 \supseteq O_2$ for all $O_1, O_2 \in \wp(\text{ALoc})$, and $(\mathcal{T}_i) \to \mathcal{T} \supseteq^* (\mathcal{T}_i') \to \mathcal{T}'$ iff $\mathcal{T}_i \supseteq^* \mathcal{T}_i'$ and $\mathcal{T}' \supseteq^* \mathcal{T}$.*

A substitution S : TVar $\to$ PType *solves a constraint $\mathcal{T}_1 \supseteq \mathcal{T}_2$ if it is a variable assignment and $[\![T]\!]\ S \supseteq^* [\![T]\!]\ S$.* □

Notice that a function type is contra-variant in the result type. The set of solutions to a constraint system $\mathcal{C}$ is denoted by Sol($\mathcal{C}$). The constraint systems we will consider all have at least one solution.

Order solutions by subset inclusion. Then a constraint system has a minimal solution, which is a "most" accurate solution to the pointer analysis problem.

4.5.2 Constraint generation

We give a constraint-based formulation of the pointer analysis specification from the previous section.

Definition 4.7 *Let $p = \langle \mathcal{T}, \mathcal{D}, \mathcal{F} \rangle$ be a program. The pointer-analysis constraint system $\mathcal{C}_{pgm}(p)$ for p is defined by*

$$\mathcal{C}_{pgm}(p) = \bigcup_{t \in \mathcal{T}} \mathcal{C}_{tdef}(t) \cup \bigcup_{d \in \mathcal{D}} \mathcal{C}_{decl}(d) \cup \bigcup_{f \in \mathcal{F}} \mathcal{C}_{fun}(f) \cup \mathcal{C}_{goal}(p)$$

where the constraint generating functions are defined below. □

Below we implicitly assume that the constraint $T_{unknown} \supseteq \{\text{Unknown}\}$ is included in all constraint systems. It implements Condition 5 in Definition 4.3 of pointer abstraction.

Goal parameters

Recall that we assume that only a "goal" function is called from the outside. The content of the goal function's parameters is unknown. Hence, we define

$$\mathcal{C}_{goal}(p) = \bigcup \{T_x \supseteq \{\text{Unknown}\}\}$$

for the goal parameters $x : T$ of the goal function in p.

Example 4.20 For the main function '`int main(int argc, char **argv)`' we have:

$$\mathcal{C}_{goal} = \{T_{argc} \supseteq \{\text{Unknown}\}, T_{argv} \supseteq \{\text{Unknown}\}\}$$

since the content of both is unknown at program start-up. **End of Example**

Declaration

Let $d \in \text{Decl}$ be a declaration. The constraint system $\mathcal{C}_{decl}(d)$ for d is defined by Figure 37.

Lemma 4.3 *Let $d \in \text{Decl}$ be a declaration. Then $\mathcal{C}_{decl}(d)$ has a solution S, and*

$$S_{|\text{ALoc}} \vdash^{pdecl} d : \bullet$$

where $\vdash^{pdecl}$ is defined by Figure 35.

Proof To see that the constraint system $\mathcal{C}_{decl}(d)$ has a solution, observe that the trivial substitution S_{triv} is a solution.

It is easy to see that a solution to the constraint system is a pointer abstraction, cf. proof of Lemma 4.1. □

[decl]	$\dfrac{\vdash^{ctype} \langle T, x\rangle : T_t}{\vdash^{cdecl}\ x : T\ :\ T_x}$	$\{T_x \supseteq T_t\}$	$\dfrac{\vdash^{cetype} \langle T, x\rangle : T_t}{\vdash^{cdecl}\ \texttt{extern}\ x : T\ :\ T_x}$	$\{T_x \supseteq T_t\}$
[base]	$\vdash^{ctype} \langle\langle\tau_b\rangle, l\rangle : T$	$\{T \supseteq \{\text{Unknown}\}\}$	$\vdash^{cetype} \langle\langle\tau_b\rangle, l\rangle : T$	$\{T \supseteq \{\text{Unknown}\}\}$
[struct]	$\vdash^{ctype} \langle\langle\text{struct S}\rangle, l\rangle : T$	$\{T \supseteq \{\}\}$	$\vdash^{cetype} \langle\langle\text{struct S}\rangle, l\rangle : T$	$\{T \supseteq \{\text{Unknown}\}\}$
[union]	$\vdash^{ctype} \langle\langle\text{union U}\rangle, l\rangle : T$	$\{T \supseteq \{\}\}$	$\vdash^{cetype} \langle\langle\text{union U}\rangle, l\rangle : T$	$\{T \supseteq \{\text{Unknown}\}\}$
[ptr]	$\vdash^{ctype} \langle\langle*\rangle T', l\rangle : T$		$\vdash^{cetype} \langle\langle*\rangle T', l\rangle : T$	
[array]	$\dfrac{\vdash^{ctype} \langle T', l[]\rangle : T_1}{\vdash^{ctype} \langle\langle[n]\rangle T', l\rangle : T}$	$\{T \supseteq \{l[]\}\}$	$\dfrac{\vdash^{cetype} \langle T', l[]\rangle : T_1}{\vdash^{cetype} \langle[n]T', l\rangle : T}$	$\{T \supseteq \{l[]\}\}$
[fun]	$\dfrac{\vdash^{cdecl} d_i : T_{d_i}}{\vdash^{ctype} \langle\langle(d_i)\rangle T', l\rangle : T}$		$\dfrac{\vdash^{cdecl} d_i : T_{d_i}}{\vdash^{cetype} \langle\langle(d_i)\rangle T', l\rangle : T}$	

Figure 37: Constraint generation for declarations

[struct]	$\dfrac{\vdash^{cdecl} d_i : T}{\vdash^{ctdef}\ \texttt{struct}\ S\{\ d_i\ \} : \bullet}$
[union]	$\dfrac{\vdash^{cdecl} d_i : T}{\vdash^{ctdef}\ \texttt{union}\ U\{\ d_i\ \} : \bullet}$
[enum]	$\vdash^{ctdef}\ \texttt{enum}\ E\{e\} : \bullet$

Figure 38: Constraint generation for type definitions

Type definitions

The constraint generation for a type definition t, $\mathcal{C}_{tdef}(t)$, is shown in Figure 38.

Lemma 4.4 *Let $t \in$ TDef be a type definition. Then the constraint system $\mathcal{C}_{tdef}(t)$ has a solution S, and it is a pointer abstraction with respect to t.*

Proof Follows from Lemma 4.3. □

Example 4.21 To implement sharing of common initial members of unions, a suitable number of inclusion constraints are added to the constraint system. **End of Example**

Expressions

Let e be an expression in a function f. The constraint system $\mathcal{C}_{exp}(e)$ for e is defined by Figure 39.

The constraint generating function O_c for operators is defined similarly to O used in the specification for expressions. We omit a formal definition.

Example 4.22 For the application '`p - q`', where '`p`' and '`q`' are pointers, we have $O_c(-_{*int,*int}, T_e, T_{e_i}) = \{T_e \supseteq \{\text{Unknown}\}\}$. In the case of an application '`p - 1`', we have $O_c(-_{*int,int}, T_e, T_{e_i}) = \{T_e \supseteq T_{e_1}\}$, cf. Example 4.17. **End of Example**

[const]	$\vdash^{cexp} c : T_e$	$\{T_e \supseteq \{\text{Unknown}\}\}$
[string]	$\vdash^{cexp} s : T_e$	$\{T_e \supseteq \{s\}\}$
[var]	$\vdash^{cexp} v : T_e$	$\{T_e \supseteq \{v\}\}$
[struct]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} e_1.i : T_e}$	$\{T_e \supseteq T_{e_1}.i\}$
[indr]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} *e_1 : T_e}$	$\{T_e \supseteq *T_{e_1}\}$
[array]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} e_1\texttt{[}e_2\texttt{]} : T_e}$	$\{T_e \supseteq *T_{e_1}\}$
[addr]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} \&^l e_1 : T_e}$	$\{T_e \supseteq \{l\}, T_l \supseteq T_e\}$
[unary]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} o\ e_1 : T_e}$	$O_c(o, T_e, T_{e_1})$
[binary]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} e_1\ o\ e_2 : T_e}$	$O_c(o, T_e, T_{e_i})$
[ecall]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} ef(e_1, \ldots, e_n)}$	$\{T_e \supseteq \{\text{Unknown}\}\}$
[alloc]	$\vdash^{cexp} \texttt{alloc}^l\texttt{(}T\texttt{)} : T_e$	$\{T_e \supseteq \{T_l\}\}$
[user]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} f^l(e_1, \ldots, e_n) : T_e}$	$\{*\{f\} \supseteq (*T_{e_i}) \to T_l, T_e \supseteq \{l\}\}$
[call]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} e_0^l(e_1, \ldots, e_n) : T_e}$	$\{*T_{e_0} \supseteq (*T_{e_i}) \to T_l, T_e \supseteq \{l\}\}$
[pre]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} \texttt{++}e_1 : T_e}$	$\{T_e \supseteq T_{e_1}\}$
[post]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} e_1\texttt{++} : T_e}$	$\{T_e \supseteq T_{e_1}\}$
[assign]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} e_1\ aop\ e_2 : T_e}$	$\{*T_{e_1} \supseteq *T_{e_2}, T_e \supseteq T_{e_2}\}$
[comma]	$\dfrac{\vdash^{cexp} e_i : T_{e_i}}{\vdash^{cexp} e_1, e_2 : T_e}$	$\{T_e \supseteq T_{e_2}\}$
[sizeof]	$\vdash^{cexp} \texttt{sizeof(}T\texttt{)} : T_e$	$\{T_e \supseteq \{\text{Unknown}\}\}$
[cast]	$\dfrac{\vdash^{cexp} e_1 : T_{e_1}}{\vdash^{cexp} (T)e_1 : T_e}$	$\text{Cast}_c(T, \text{TypOf}(e_1), T_e, T_{e_1})$

Figure 39: Constraint generation for expressions

To represent the lvalue of the result of a function application, we use a "fresh" variable T_l. For reasons to be seen in the next section, calls are assumed to be labeled.

The function Cast_c implementing constraint generation for casts is defined as follows.

$$\begin{array}{lcl}
\mathrm{Cast}_c(T_{to}, T_{from}, T_e, T_{e_1}) & = & \text{case } (T_{to}, T_{from}) \text{ of} \\
(\langle\tau_b\rangle, \langle\tau_b\rangle) & : & \{T_e \supseteq T_{e_1}\} \\
(\langle *\rangle\, T, \langle\tau_b\rangle) & : & \{T_e \supseteq \{\text{Unknown}\}\} \\
(\langle\tau_b\rangle, \langle *\rangle\, T) & : & \{T_e \supseteq \{\text{Unknown}\}\} \\
(\langle *\rangle\, T, \langle *\rangle\, \langle\text{struct S}\rangle) & : & \begin{cases} \{T_e \supseteq T_{e_1}.1\} & T \text{ type of first member of } S \\ \{T_e \supseteq T_{e_1}\} & \text{Otherwise} \end{cases} \\
(\langle *\rangle\, T_1, \langle *\rangle\, T_2) & : & \{T_e \supseteq T_{e_1}\}
\end{array}$$

Notice the resemblance with function Cast defined in Section 4.4.

Lemma 4.5 *Let $e \in \mathrm{Expr}$ be an expression. Then $\mathcal{C}_{exp}(e)$ has a solution S, and*

$$S_{|\mathrm{ALoc}} \vdash^{pexp} e : V$$

where $\vdash^{pexp}$ is defined by Figure 36.

Proof To see that $\mathcal{C}_{exp}(e)$ has a solution, observe that S_{triv} is a solution.

That S is a safe pointer abstraction for e follows from definition of pointer types (Definition 4.5) and solution to constraint systems (Definition 4.6). □

Example 4.23 Consider the call '`f`^{1}`(&`^{2}`x)`'; a (simplified) constraint system is

$$\{T_{\&x} \supseteq \{2\}, T_2 \supseteq \{x\}, T_f \supseteq \{f\}, *T_f \supseteq (*T_{\&x}) \to T_1, T_{f()} \supseteq \{1\}\}$$

cf. Figure 39. By "natural" rewritings (see Section 4.7) we get

$$\{(T_{f_1}) \to T_{f_0} \supseteq (*\{2\}) \to T_1, T_{f()} \supseteq \{1\}\}$$

(where we have used that T_f is bound to $(T_{f_1}) \to T_{f_0}$) which can be rewritten to

$$\{(T_{f_1}) \to T_{f_0} \supseteq (T_2) \to T_1, T_{f()} \supseteq \{1\}\}$$

(where we have used that $*\{2\} \Rightarrow T_2$) corresponding to

$$\{T_{f_1} \supseteq \{x\}, T_{f()} \supseteq T_1\}$$

that is, the parameter of f may point to '`x`', and f may return the value in location '1'. Notice that use of contra-variant in the last step. **End of Example**

$$
\begin{array}{lll}
\text{[empty]} & \vdash^{cstmt} ; : \bullet & \\[4pt]
\text{[expr]} & \dfrac{\vdash^{cexp} e : T_e}{\vdash^{cstmt} e : \bullet} & \\[4pt]
\text{[if]} & \dfrac{\vdash^{cexp} e : T_e \quad \vdash^{cstmt} S_i : \bullet}{\vdash^{cstmt} \texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2 : \bullet} & \\[4pt]
\text{[switch]} & \dfrac{\vdash^{cexp} e : T_e \quad \vdash^{cstmt} S_1 : \bullet}{\vdash^{cstmt} \texttt{switch (}e\texttt{)}\ S_1 : \bullet} & \\[4pt]
\text{[case]} & \dfrac{\vdash^{cstmt} S_1 : \bullet}{\vdash^{cstmt} \texttt{case}\ e\texttt{:}\ S_1 : \bullet} & \\[4pt]
\text{[default]} & \dfrac{\vdash^{cstmt} S_1 : \bullet}{\vdash^{cstmt} \texttt{default}\ S_1 : \bullet} & \\[4pt]
\text{[while]} & \dfrac{\vdash^{cexp} e : T_e \quad \vdash^{cstmt} S : \bullet}{\vdash^{cstmt} \texttt{while (}e\texttt{)}\ S_1 : \bullet} & \\[4pt]
\text{[do]} & \dfrac{\vdash^{cexp} e : T_e \quad \vdash^{cstmt} S_1 : \bullet}{\vdash^{csmt} \texttt{do}\ S_1\ \texttt{while (}e\texttt{)} : \bullet} & \\[4pt]
\text{[for]} & \dfrac{\vdash^{cexp} e_i : T_{e_i} \quad \vdash^{cstmt} S_1 : \bullet}{\vdash^{cstmt} \texttt{for(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{)}\ S_1 : \bullet} & \\[4pt]
\text{[label]} & \dfrac{\vdash^{cstmt} S_1 : \bullet}{\vdash^{cstmt} l\texttt{:}\ S_1 : \bullet} & \\[4pt]
\text{[goto]} & \vdash^{cstmt} \texttt{goto}\ m : \bullet & \\[4pt]
\text{[return]} & \dfrac{\vdash^{cexp} e : T_e}{\vdash^{cstmt} \texttt{return}\ e : \bullet} & \{T_{f_0} \supseteq *T_e\} \\[4pt]
\text{[block]} & \dfrac{\vdash^{cstmt} S_i : \bullet}{\vdash^{cstmt} \{S_i\} : \bullet} &
\end{array}
$$

Figure 40: Constraint generation for statements

Statements

Suppose $s \in \text{Stmt}$ is a statement in a function f. The constraint system $\mathcal{C}_{stmt}(s)$ for s is defined by Figure 40.

The rules basically collect the constraints for contained expressions, and add a constraint for the `return` statement.

Lemma 4.6 *Let $s \in \text{Stmt}$ be a statement in function f. Then $\mathcal{C}_{stmt}(s)$ has a solution S, and S is a safe pointer abstraction for s.*

Proof Follows from Lemma 4.5. □

Functions

Let $f \in \text{Fun}$ be a function definition $f = \langle T, \mathcal{D}_{par}, \mathcal{D}_{loc}, \mathcal{S}\rangle$. Define

$$\mathcal{C}_{fun}(f) = \bigcup_{d \in \mathcal{D}_{par}} \mathcal{C}_{decl}(d) \cup \bigcup_{d \in \mathcal{D}_{loc}} \mathcal{C}_{decl}(d) \cup \bigcup_{s \in \mathcal{S}} \mathcal{C}_{stmt}(s)$$

where $\mathcal{C}_{decl}$ and $\mathcal{C}_{stmt}$ are defined above.

Lemma 4.7 *Let $f \in \mathrm{Fun}$ be a function. Then $\mathcal{C}_{fun}(f)$ has a solution S, and S is a safe pointer abstraction for f.*

Proof Obvious. □

This completes the specification of constraint generation.

4.5.3 Completeness and soundness

Given a program p. We show that $\mathcal{C}_{pgm}$ has a solution and that the solution is a safe pointer abstraction.

Lemma 4.8 *Let p be a program. The constraint system $\mathcal{C}_{pgm}(p)$ has a solution.*

Proof The trivial solution S_{triv} solves $\mathcal{C}_{pgm}(p)$. □

Theorem 4.1 *Let p be a program. A solution $S \in \mathrm{Sol}(\mathcal{C}_{pgm}(p))$ is a* safe pointer abstraction *for p.*

Proof Follows from Lemma 4.7, Lemma 4.3 and Lemma 4.4. □

4.6 Inter-procedural pointer analysis

The intra-procedural analysis developed in the previous section sacrifices accuracy at functions calls: *all* calls to a function are merged. Consider for an example the following function:

```
/* inc_ptr: increment pointer p */
int *inc_ptr(int *q)
{
   return q + 1;
}
```

and suppose there are two calls '`inc_ptr(a)`' and '`inc_ptr(b)`', where '`a`' and '`b`' are pointers. The intra-procedural analysis merges the calls and alleges a call to '`inc_ptr`' yields a pointer to either '`a`' or '`b`'

With many calls to '`inc_ptr()`' spurious point-to information is propagated to unrelated call-sites, degrading the accuracy of the analysis. This section remedies the problem by extending the analysis into an *inter-procedural*, or *context-sensitive* point-to analysis.

4.6.1 Separating function contexts

The naive approach to inter-procedural analysis is by textual copying of functions before intra-procedural analysis. Functions called from different contexts are copied, and the call-sites changed accordingly. Copying may increase the size of the program exponentially, and henceforth also the generated constraint systems.

Example 4.24 Consider the following program.

```
int main(void)                int *dinc(int *p)
{                             {
   int *pa,*pb,a[10],b[10];      int *p1 = inc_ptr(p);
   px = dinc(a);                 int *p2 = int_ptr(p1);
   py = dinc(b);                 return p2;
}                             }
```

Copying of function '`dinc()`' due to the two calls in '`main()`' will create two variants with 4 calls to '`int_ptr()`'. **End of Example**

The problem with textual copying of functions is that the analysis is slowed down due to the increased number of constraints, and worse, the copying may be useless: copies of function may be used in "similar" contexts such that copying does not enhance accuracy. Ideally, the cloning of functions should be based on the *result* of the analysis, such that only functions that gain from copying actually are copied.

Example 4.25 The solution to intra-procedural analysis of Example 4.24 is given below.

$$\begin{array}{lll} T_{pa} & \mapsto & \{a,b\} \\ T_{pb} & \mapsto & \{a,b\} \\ T_{p} & \mapsto & \{a,b\} \\ T_{q} & \mapsto & \{a,b\} \end{array}$$

where the calls to '`dinc()`' have been collapsed. By copying of '`inc_ptr()`' four times, the pointers '`a`' and '`b`' would not be mixed up. **End of Example**

4.6.2 Context separation via static-call graphs

We employ the program's static-call graph to differentiate functions in different contexts. Recall that a program's static-call graph is a function $\mathcal{SCG}$: CallLabel $\times$ Variant $\rightarrow$ Id $\times$ Variant mapping a call-site and a variant number of the enclosing function to a function name and a variant. The static-call graph of the program in Example 4.24 is shown in Figure 41. Four variants of '`inc_ptr()`' exist due to the two call-sites in '`dinc()`' which again is called twice from '`main()`'.

Explicit copying of functions amounts to creating the variants as indicated by Figure 41. However, observe: the constraint systems generated for the variants are *identical* except for constraints for calls and `return`. The idea is to generate constraints over *vectors* of pointer types corresponding to the number of variants. For example, the constraint

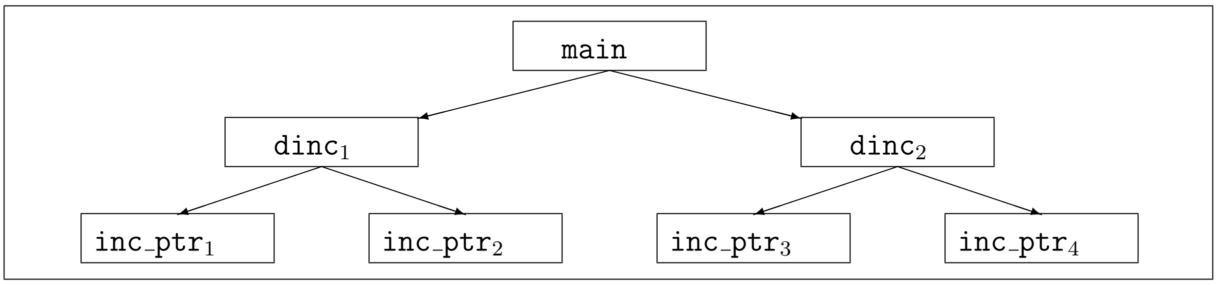

Figure 41: Static-call graph for the example program

system for '`inc_ptr()`' will use vectors of length 5, since there are four variants. Variant 0 is used as a *summary* variant, and for indirect calls.

After the analysis, procedure cloning can be accomplished on the basis of the computed pointer information. Insignificant variants can be eliminated and replaced with more general variants, or possibly with the summary variant 0.

4.6.3 Constraints over variant vectors

Let an *extended constraint system* be a multi-set of extended constraints

$$\mathcal{T}^n \supseteq \mathcal{T}^n$$

where $\mathcal{T}$ range over pointer types. Satisfiability of constraints is defined by component-wise extension of Definition 4.6.

Instead of assigning a single type variable to objects and expressions, we assign a *vector* $\overline{T}$ of type variables. The length is given as the number of variants of the encapsulating function (plus the 0 variant) or 1 in the case of global objects.

Example 4.26 Consider again the program in Example 4.24. Variable '`p`' of '`dinc`' is associated with the vector $p \mapsto \left\langle T_p^0, T_p^1, T_p^2 \right\rangle$ corresponding to variant 1 and 2, and the summary 0. The vector corresponding to the parameter of '`inc_ptr()`' has five elements due to the four variants. **End of Example**

The vector of variables associated with object o is denoted by $\overline{T_o} = \langle T_o^0, T_o^1, \ldots, T_o^n \rangle$. Similarly for expressions and types.

Example 4.27 An inter-procedural solution to the pointer analysis problem in Example 4.24:

$$\begin{array}{lll}
\left\langle T_{pa}^0, T_{pa}^1 \right\rangle & \mapsto & \langle \{a\}, \{a\} \rangle \\
\left\langle T_{pb}^0, T_{pb}^1 \right\rangle & \mapsto & \langle \{b\}, \{b\} \rangle \\
\left\langle T_p^0, T_p^1, T_p^2 \right\rangle & \mapsto & \langle \{a,b\}, \{a\}, \{b\} \rangle \\
\left\langle T_q^0, T_q^1, T_q^2, T_q^3, T_q^4 \right\rangle & \mapsto & \langle \{a,b\}, \{a\}, \{a\}, \{b\}, \{b\} \rangle
\end{array}$$

where the context numbering is shown in Figure 41. **End of Example**

In the example above, it would be advantageous to merge variant 1, 2, and 3, 4, respectively.

4.6.4 Inter-procedural constraint generation

The inter-procedural constraint generation proceeds almost as in the intra-procedural analysis, Section 4.5.2, except in the cases of calls and `return` statements. Consider constraint generation in a function with n variants.

The rule for constants:

$$\{\overline{T_e} \supseteq \langle\{\text{Unknown}\},\{\text{Unknown}\},\dots,\{\text{Unknown}\}\rangle\}$$

where the length of the vector is $n+1$. The rule for variable references:

$$\begin{array}{ll} \{\overline{T_e} \supseteq \langle\{v\},\{v\},\dots,\{v\}\rangle\} & \text{if } v \text{ is global} \\ \{\overline{T_e} \supseteq \langle\{v^0\},\{v^1\},\dots,\{v^n\}\rangle\} & \text{if } v \text{ is local} \end{array}$$

where v^i denote the i'th variant of object v. This rule seems to imply that there exists n versions of v. We describe a realization below. (The idea is that an object is uniquely identified by its associated variable, so in practice the rule reads $\overline{T_e} \supseteq \langle\{T_v^0\},\{T_v^1\},\dots,\{T_v^n\}\rangle$.)

Consider a call $g^l(e_1,\dots,e_m)$ in function f. The constraint system is

$$\bigcup_{i=1,\dots,n}\{T^{k^i}_{g_j} \supseteq *T^i_{e_j}\} \cup \bigcup_{u=1,\dots,n}\{T_{l^i} \supseteq T^{k^i}_{g_0}\} \cup \{\overline{T_e} \supseteq \overline{\{l^i\}}\}$$

where $\mathcal{SCG}(l,i) = \langle g, k^i\rangle$.

The rule is justified as follows. The i'th variant of the actual parameters are related to the corresponding variant k^i of the formal parameters, cf. $\mathcal{SCG}(l,i) = \langle g, k^i\rangle$. Similarly for the result. The abstract location l abstracts the lvalue(s) of the call.

The rule for an indirect call $e_0(e_1,\dots,e_n)$ uses the summary nodes:

$$\{*T^0_{e_0} \supseteq (*T^0_{e_i}) \to T^0_l, \overline{T_e} \supseteq \Big\langle\{l^0\},\{l^0\},\dots,\{l^0\}\Big\rangle\}$$

cf. the rule for intra-procedural analysis. Thus, no context-sensitivity is maintained by indirect calls.

Finally, for every definition '$x : T$' that appears in n variants, the constraints

$$\bigcup_{i=1,\dots,n}\{T^0_x \supseteq T^i_x\}$$

are added. This assures that variant 0 of a type vector summarizes the variants.

Example 4.28 The first call to '`inc_ptr()`' in Example 4.24 gives rise to the following constraints.

$$\begin{array}{ll} T^1_{inc\_ptr} \supseteq T^1_{dinc}, T^1_p \supseteq T^1_{dinc_0} & \text{variant 1} \\ T^2_{inc\_ptr} \supseteq T^2_{dinc}, T^2_p \supseteq T^2_{dinc_0} & \text{variant 2} \end{array}$$

where we for the sake of presentation have omitted "intermediate" variables, and rewritten the constraints slightly. **End of Example**

A constraint system for inter-procedural analysis consists of only a few more constraints than in the case of intra-procedural analysis. This does not mean, naturally, that a inter-procedural solution can be found in the same time as an intra-procedural solution: the processing of each constraint takes more time. The thesis is the processing of an extended constraint takes less time than processing of an increased number of constraints.

4.6.5 Improved naming convention

As a side-efect, the inter-procedural analysis improves on the accuracy with respect to heap-allocated objects. Recall that objects allocated from the same call-site are collapsed.

The constraint generation in the inter-procedural analysis for '$\texttt{alloc}^l\texttt{()}$' calls is

$$\{\overline{T_e} \supseteq \left\langle \{l^0\}, \{l^1\}, \ldots, \{l^n\} \right\rangle\}$$

where l_i are $n+1$ "fresh" variables.

Example 4.29 An intra-procedural analysis merges the objects allocated in the program below even though they are unrelated.

```
int main(void)                      struct S *allocate(void)
{                                   {
   struct S *s = allocate();           return alloc¹(S);
   struct S *t = allocate();        }
}
```

The inter-procedural analysis creates two variants of '`allocate()`', and separates apart the two invocations. **End of Example**

This gives the analysis the same accuracy with respect to heap-allocated objects as other analyses, *e.g.* various invocations of a function is distinguished [Choi *et al.* 1993].

4.7 Constraint solving

This section presents a set of solution-preserving rewrite rules for constraint systems. We show that repeated application of the rewrite rules brings the system into a form where a solution can be found easily. We argue that this solution is minimal.

For simplicity we consider intra-procedural constraints only in this section. The extension to inter-procedural systems is straightforward: pairs of types are processed component-wise. Notice that the same number of type variables always appear on both sides of a constraint. In practice, a constraint is annotated with the length of the type vectors.

4.7.1 Rewrite rules

Let $\mathcal{C}$ be a constraint system. The application of rewrite rule l resulting in system $\mathcal{C}'$ is denoted by $\mathcal{C} \Rightarrow^l \mathcal{C}'$. Repeated application of rewrite rules is written $\mathcal{C} \Rightarrow \mathcal{C}'$. Exhausted application[13] is denoted by $\mathcal{C} \Rightarrow^* \mathcal{C}'$ (we see below that exhausted application makes sense).

A rewrite rule l is *solution preserving* if a substitution S is a solution to $\mathcal{C}$ if and only if it is a solution to $\mathcal{C}'$, when $\mathcal{C} \Rightarrow^l \mathcal{C}'$. The aim of constraint rewriting is to propagate point-to sets through the type variables. The rules are presented in Figure 42, and make use of an auxiliary function Collect : TVar $\times$ CSystem $\rightarrow \wp(\text{ALoc})$ defined as follows.

[13]Application until the system stabilizes.

Type normalization			
1.a	$\mathcal{C} \equiv \mathcal{C}' \cup \{T \supseteq \{s\}.i\}$	$\Rightarrow$	$\mathcal{C} \cup \{T \supseteq T_{S_i}\}$ $\quad \mathrm{TypOf}(s) = \langle \text{struct S} \rangle$
1.b	$\mathcal{C} \equiv \mathcal{C}' \cup \{T \supseteq *\{o\}\}$	$\Rightarrow$	$\mathcal{C} \cup \{T \supseteq T_o\}$
1.c	$\mathcal{C} \equiv \mathcal{C}' \cup \{*\{o\} \supseteq \mathcal{T}\}$	$\Rightarrow$	$\mathcal{C} \cup \{T_0 \supseteq \mathcal{T}\}$ $\quad o \mapsto T_o$
1.d	$\mathcal{C} \equiv \mathcal{C}' \cup \{(T_i) \to T \supseteq (\mathcal{T}_i') \to T'\}$	$\Rightarrow$	$\mathcal{C} \cup \{T_i \supseteq \mathcal{T}_i', T' \supseteq T\}$
Propagation			
2.a	$\mathcal{C} \equiv \mathcal{C}' \cup \{T_1 \supseteq T_2\}$	$\Rightarrow$	$\mathcal{C} \cup \bigcup_{o \in \mathrm{Collect}(T_2, \mathcal{C})} \{T_1 \supseteq \{o\}\}$
2.b	$\mathcal{C} \equiv \mathcal{C}' \cup \{T_1 \supseteq T_2.i\}$	$\Rightarrow$	$\mathcal{C} \cup \bigcup_{o \in \mathrm{Collect}(T_2, \mathcal{C})} \{T \supseteq \{o\}.i\}$
2.c	$\mathcal{C} \equiv \mathcal{C}' \cup \{T_1 \supseteq *T_2\}$	$\Rightarrow$	$\mathcal{C} \cup \bigcup_{o \in \mathrm{Collect}(T_2, \mathcal{C})} \{T_1 \supseteq *\{o\}\}$
2.d	$\mathcal{C} \equiv \mathcal{C}' \cup \{*T \supseteq \mathcal{T}\}$	$\Rightarrow$	$\mathcal{C} \cup \bigcup_{o \in \mathrm{Collect}(T, \mathcal{C})} \{*\{o\} \supseteq \mathcal{T}\}$

Figure 42: Solution preserving rewrite rules

Definition 4.8 *Let $\mathcal{C}$ be a constraint system. The function* Collect : TVar $\times$ CSystem $\to$ $\wp$(ALoc) *is defined inductively by:*

$$\mathrm{Collect}(T, \mathcal{C}) = \{o_i \mid T \supseteq \{o_i\} \in \mathcal{C}\} \cup \{o \mid T \supseteq T_1 \in \mathcal{C}, o_i \in \mathrm{Collect}(T_1, \mathcal{C})\}$$

$\Box$

Notice that constraints may be self-dependent, *e.g.* a constraint system may contain constraints $\{T_1 \supseteq T_2, T_2 \supseteq T_1\}$.

Lemma 4.9 *Let $\mathcal{C}$ be a constraint system and suppose that T is a variable appearing in $\mathcal{T}$. Then* $\mathrm{Sol}(\mathcal{C}) = \mathrm{Sol}(\mathcal{C} \cup \{T \supseteq \mathrm{Collect}(T, \mathcal{C})\})$.

Proof Obvious. $\Box$

For simplicity we have assumed abstract location sets $\{o\}$ consist of one element only. The generalization is straightforward. Constraints of the form $\{o\}.i \supseteq \mathcal{T}$ can never occur; hence no rewrite rule.

Lemma 4.10 *The rules in Figure 42 are solution preserving.*

Proof Assume that $\mathcal{C}_l \Rightarrow^l \mathcal{C}_r$. We show: S is a solution to $\mathcal{C}$ iff it is a solution to $\mathcal{C}'$.
Cases 1: The rules follow from the definition of pointer types (Definition 4.5). Observe that due to static well-typedness, "s" in rule 1.a denotes a struct object.
Case 2.a: Due to Lemma 4.9.
Case 2.b: Suppose that S is a solution to $\mathcal{C}_l$. By Lemma 4.9 and definition of pointer types, S is a solution to $\mathcal{C}_l \cup \{T_1 \supseteq \{o\}.i\}$ for $o \in \mathrm{Collect}(T_2, \mathcal{C}_l)$. Suppose that S' is a solution to $\mathcal{C}_r$. By Lemma 4.9, S' is a solution to $\mathcal{C}_r \cup \{T_2 \supseteq \{o\}\}$ for $o \in \mathrm{Collect}(T_2, \mathcal{C}_r)$.
Case 2.c: Similar to case 2.b.
Case 2.d: Similar to case 2.b. $\Box$

Lemma 4.11 *Consider a constraint system to be a* set *of constraint. Repeated application of the rewrite rules in Figure 42 $\mathcal{C} \Rightarrow \mathcal{C}'$ terminates.*

Proof All rules add constraints to the system. This can only be done a finite number of times. $\Box$

Thus, when considered as a set, a constraint system $\mathcal{C}$ has a *normal form* $\mathcal{C}'$ which can be found by exhaustive application $\mathcal{C} \Rightarrow \mathcal{C}'$ of the rewrite rules in Figure 42.

Constraint systems in normal form have a desirable property: a solution can be found directly.

4.7.2 Minimal solutions

The proof of the following theorem gives a constructive (though inefficient) method for finding a minimal solution to a constraint system.

Theorem 4.2 *Let $\mathcal{C}$ be a constraint system. Perform the following steps:*

1. *Apply the rewrite rules in Figure 42 until the system stabilizes as system $\mathcal{C}'$.*
2. *Remove from $\mathcal{C}'$ all constraints but constraints of the form $T \supseteq \{o\}$ giving $\mathcal{C}''$.*
3. *Define the substitution S by $S = [T \mapsto \mathrm{Collect}(T, \mathcal{C}'')]$ for all T in $\mathcal{C}''$.*

Then $S_{|\mathrm{ALoc}} \in Sol(\mathcal{C})$, and S is a minimal solution.

Proof Due to Lemma 4.10 and Lemma 4.9 it suffices to show that S is a solution to $\mathcal{C}'$.

Suppose that S is not a solution to $\mathcal{C}'$. Clearly, S is a solution to the constraints added during rewriting: constraints generated by rule 2.b are solved by 1.a, 2.c by 1.b, and 2.d by 1.c. Then there exists a constraint $c \in \mathcal{C} \setminus \mathcal{C}'$ which is not satisfied. Case analysis:

- $c = T_1 \supseteq \{o\}$: Impossible due to Lemma 4.9.
- $c = T_1 \supseteq T_2$: Impossible due to exhaustive application of rule 2.a and Lemma 4.9.
- $c = T_1 \supseteq T_2.i$: Impossible due to rewrite rule 2.b and Lemma 4.9.
- $c = T_1 \supseteq *T_2$: Impossible due to rewrite rule 2.c and Lemma 4.9.
- $c = *T_1 \supseteq \mathcal{T}$: Impossible due to rewrite rules 2.d and 1.c, and Lemma 4.9.

Hence, S is a solution to $\mathcal{C}'$.

To see that S is minimal, notice that no inclusion constraints $T_1 \supseteq \{o\}$ than needed are added; thus S must be a minimal solution. $\Box$

The next section develops an iterative algorithm for pointer analysis.

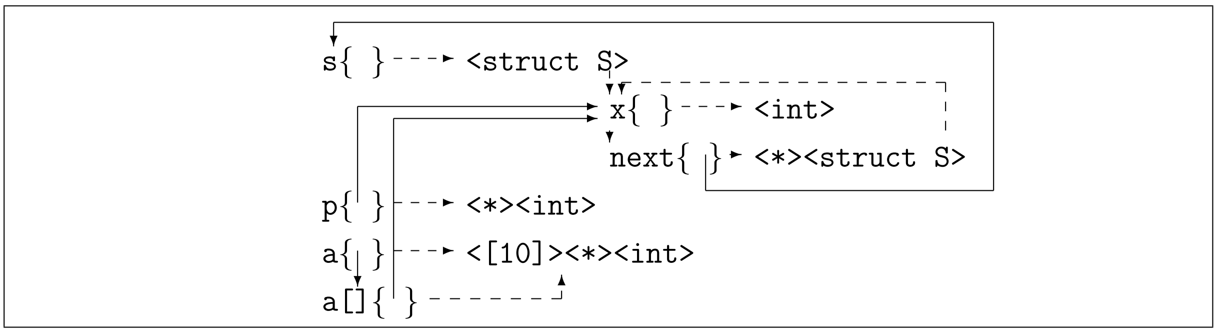

Figure 43: Pointer type representation

4.8 Algorithm aspects

In this section we outline an algorithm for pointer analysis. The algorithm is similar to classical iterative fixed-point solvers [Aho *et al.* 1986,Kildall 1973]. Further, we describe a convenient representation.

4.8.1 Representation

To every declarator in the program we associate a pointer type. For abstract locations that do not have a declarator, *e.g.* $a[]$ in the case of an array definition '`int a[10]`', we create one. A object is uniquely identified by a pointer to the corresponding declarator. Thus, the constraint $T \supseteq *\{o\}$ is represented as $T \supseteq *\{T_o\}$ which can be rewritten into $T \supseteq T_o$ in constant time.

Example 4.30 The "solution" to the pointer analysis problem of the program below is shown in Figure 43.

```
struct S { int x; struct S *next; } s;
int *p, *a[10];
s.next = &s;
p = a[1] = &s.x;
```

The dotted lines denotes representation of static types.

End of Example

To every type variable '`T`' we associate a set '`T.incl`' of (pointers to) declarators. Moreover, a boolean flag '`T.upd`' is assumed for each type variable. The field '`T.incl`' is incrementally updated with the set of objects '`T`' includes. The flag '`T.upd`' indicates whether a set has changed since "last inspection".

4.8.2 Iterative constraint solving

Constraints of the form $T \supseteq *\{T_o\}$ can be "pre-normalized" to $T \supseteq T_o$ during constraint generation, and hence do not exists during the solving process. Similar for constraint generated for user-function call.

The constraint solving algorithm is given as Algorithm 4.1 below.

Algorithm 4.1 *Iterative constraint solving.*

```
do
   fix = 1;
   for (c in clist)
      switch (c) {
         case T1 ⊇ O: update(T1,O); break;
         case T1 ⊇ T2: update(T1,T2.incl); break;
         case T1 ⊇ T2.i: update(T1,struct(T2.incl,i)); break;
         case T1 ⊇ *T2:
            update(T1,indr(T2.incl)));
            if (Unknown in T2.incl) abort("Unknown dereferenced");
            break;
         case *T1 ⊇ *T2:
            if (T1.upd || T2.upd) {
               for (o in T1.incl)
                  update(To,indr(T2.incl));
               }
            break;
         case *T0 ⊇ (*T'i)->T':
            if (T0.upd) {
               for ((Ti)->T in T0.incl)
                  clist ∪= { Ti ⊇ *T'i, T' ⊇ T };
               }
             break;
      }
while (!fix);

/* update: update content T.incl with O */
update(T,O)
{
   if (T.incl ⊈ O) { T.incl ∪= O; fix = 0; }
}
```

Functions '`indr()`*' and '*`struct()`*' are defined the obvious way. For example, '*`indr()`*' dereferences (looks up the binding of) a declarator pointer (location name) and returns the point-to set.* □

Notice case for pointer deference. If Unknown is dereferenced, the algorithm aborts with a "worst-case" message. This is more strict than needed. For example, the analysis yields "worst-case" in the case of an assignment '`p = *q`', where '`q`' is approximated by Unknown. In practice, constraints appearing at the left hand side of assignments are "tagged", and only those give rise to abortion.

4.8.3 Correctness

Algorithm 4.1 terminates since the '`incl`' fields only can be update a finite number of times. Upon termination, the solution is given by $S = [T \mapsto \texttt{T.incl}]$.

Lemma 4.12 *Algorithm 4.1 is correct.*

Proof The algorithm implements the rewrite rules in Figure 42. □

4.8.4 Complexity

Algorithm 4.1 is polynomial in the size of program (number of declarators). It has been shown that inter-procedural may-alias in the context of multi-level pointers is P-space hard [Landi 1992a].[14] This indicates the degree of approximation our analysis make. On the other hand, it is fast and the results seem reasonable.

4.9 Experiments

We have implemented a pointer analysis in the *C-Mix* system. The analysis is similar to the one presented in this chapter, but deviates it two ways: it uses a representation which reduces the number of constraints significantly (see below), and it computes summary information for all indirections of pointer types.

The former decreases the runtime of the analysis, the latter increases it. Notice that the analysis of this chapter only computes the lvalues of the first indirection of a pointer; the other indirections must be computed by inspections of the objects to which a pointer may point.

The value of maintaining summary information for all indirections depends on the usage of the analysis. For example with summary information for all indirections, the side-effect analysis of Chapter 6 does not need to summarize pointers at every indirection node; this is done during pointer analysis. On the other hand, useless information may be accumulated. We suspect that the analysis of this chapter is more feasible in practice, but have at the time of writing no empirical evidence for this.

We have applied the analysis to some test programs. All experiments were conducted on a Sun SparcStation II with 64 Mbytes of memory. The results are shown below. We refer to Chapter 9 for a description of the programs.

Program	Lines	Constraints	Solving
Gnu strstr	64	17	$\approx$ 0.0 sec
Ludcmp	67	0	0.0 sec
Ray tracer	1020	157	0.3 sec
ERSEM	$\approx$ 5000	465	3.3 sec

As can be seen, the analysis is fast. It should be stressed, however, that none of the programs use pointers extensively. Still, we believe the analysis of this chapter will exhibit comparable run times in practice. The quality of the inferred information is good. That is, pointers are approximated accurately (modulo flow-insensitivity). In average, the points-to sets for a pointer are small.

Remark: The number of constraints reported above seems impossible! The point is that most of the superset constraints generated can be solved by equality. All of these constraints are pre-normalized, and hence the constraint system basically contains only constraints for assignments (calls) involving pointers.

[14]This has only been shown for programs exhibiting more than four levels of indirection.

4.10 Towards program-point pointer analysis

The analysis developed in this chapter is flow-insensitive: it produces a summary for the entire body of a function. This benefits efficiency at the price of precision, as illustrated by the (contrived) function to the left.

```
int foo(void)
{
   if (test) {
      p = &x;
      foobar(p);
   } else {
      p = &y;
      foobar(p);
   }
}
```

```
int bar(void)
{
   p = &x;
   foobar(p);
   p = &y;
   foobar(p);
 }
```

The analysis ignores the branch and information from one branch may influence the other. In this example the loss of accuracy is manifested by the propagation of the point-to information $[p \mapsto \{x, y\}]$ to *both* calls.

The example to the right illustrates lack of program-point specific information. A program-point specific analysis will record that '`p`' will point to '`x`' at the first call, and to '`y`' at the second call. In this section we consider program-point specific, flow-sensitive pointer analysis based on constraint solving.

4.10.1 Program point is sequence point

The aim is to compute a pointer abstraction for each program point, mapping pointers to the sets of objects they may point to at that particular program point. Normally, a program point is defined to be "between two statements", but in the case of C, the notion coincides with *sequence points* [ISO 1990, Paragraph 5.1.2.3]. At a sequence point, all side-effects between the previous and the current point shall have completed, *i.e.* the store updated, and no subsequent side-effects have taken place. Further, an object shall be accessed at most once to have its value determined. Finally, an object shall be accessed only to determine the value to be stored. The sequence points are defined in Annex C of the Standard [ISO 1990].

Example 4.31 The definition renders undefined an expression such as '`p = p++ + 1`' since '`p`' is "updated" twice between two sequence points.

Many analyses rely on programs being transformed into a simpler form, *e.g.* 'e_1 `=` e_2 `=` e_3' to 'e_2 `=` e_3`;` e_1 `=` e_2'. This introduces new sequence points and may turn an undefined expression into a defined expression, for example '`p = q = p++`'. **End of Example**

In the following we for simplicity ignore sequence points in expressions, and use the convention that if S is a statement, then m is the program immediately before S, and n is the program after. For instance, for a sequence of statements, we have $m_1 S_1 n_1 m_2 S_2 n_2 \dots m_n S_n n_n$.

4.10.2 Program-point constraint-based program analysis

This section briefly recapitulates constraint-based, or set-based program analysis of imperative language, as developed by Heintze [Heintze 1992].

To every program points m, assign a vector of type variables $\overline{T}^m$ representing the abstract store.

Example 4.32 Below the result of a program-point specific analysis is shown.

```
int main(void)
{
   int x, y, *p;
   /* 1: ⟨T_x^1, T_y^1, T_p^1⟩  ↦  ⟨{}, {}, {}⟩ */
   p = &x;
   /* 2: ⟨T_x^2, T_y^2, T_p^2⟩  ↦  ⟨{}, {}, {x}⟩ */
   p = &y;
   /* 3: ⟨T_x^3, T_y^3, T_p^3⟩  ↦  ⟨{}, {}, {y}⟩ */
   x = 4;
   /* 4: ⟨T_x^4, T_y^4, T_p^4⟩  ↦  ⟨{}, {}, {y}⟩ */

}
```

Notice that T_p^3 does not contain $\{x\}$. **End of Example**

The corresponding constraint systems resemble those introduced in Section 4.5. However, extra constraints are needed to *propagate* the abstract state through the program points. For example, at program point 4, the variable T_p^4 assumes the same value as T_p^3, since it is not updated.

Example 4.33 Let $\overline{T}^n \supseteq \overline{T}^m[x \mapsto O]$ be a short hand for $T_o^n \supseteq T_o^m$ for all o except x, and $T_x^n \supseteq O$. Then the following constraints abstracts the pointer usage in the previous example:

$$\begin{array}{ll} 2: & \overline{T}^2 \supseteq \overline{T}^1[p \mapsto \{x\}] \\ 3: & \overline{T}^3 \supseteq \overline{T}^2[p \mapsto \{y\}] \\ 4: & \overline{T}^4 \supseteq \overline{T}^3[x \mapsto \{\}] \end{array}$$

End of Example

The constraint systems can be solved by the rewrite rules in Figure 42, but unfortunately the analysis cannot cope with multi-level pointers.

4.10.3 Why Heintze's set-based analysis fails

Consider the following program fragment.

```
int x, y, *p, **q;
/* 1: ⟨T_x^1,T_y^1,T_p^1,T_q^1⟩ ↦ ⟨{},{},{},{}⟩ */
p = &x;
/* 2: ⟨T_x^2,T_y^2,T_p^2,T_q^2⟩ ↦ ⟨{},{},{x},{}⟩ */
q = &p;
/* 3: ⟨T_x^3,T_y^3,T_p^3,T_q^3⟩ ↦ ⟨{},{},{x},{p}⟩ */
*q = &y;
/* 4: ⟨T_x^4,T_y^4,T_p^4,T_q^4⟩ ↦ ⟨{},{},{y},{p}⟩ */
```

The assignment between program point 3 and 4 updates the abstract location p, but 'p' does not occur syntactically in the expression '*q = &y'. Generating the constraints $\overline{T}^4 \supseteq \overline{T}^3[*T_q^3 \mapsto \{y\}]$ will incorrectly leads to $T_p^4 \supseteq \{x, y\}$.

There are two problems. First, the values to be propagated through states are not *syntactically* given by an expression, *e.g.* that 'p' will be updated between program points 3 and 4. Secondly, the indirect assignment will be modeled by a constraint of the form $*T_q^4 \supseteq \{y\}$ saying that the indirection of 'q' (that is, 'p') should be updated to contain 'y'. However, given $T_q^4 \mapsto \{p\}$, it is not apparent from the constraint that $*T_q^4 \supseteq \{y\}$ should be rewritten to $T_p^4 \supseteq \{y\}$; program points are not a part of a constraint (how is the "right" type variable for 'p' chosen?).

To solve the latter problem, constraints generated due to assignments can be equipped with program points: $\mathcal{T}^n \supseteq^m \mathcal{T}$ meaning that program point n is updated from state m. For example, $*T_q^4\ {}^4\!\supseteq^3 \{y\}$ would be rewritten to $T_p^4 \supseteq \{y\}$, since $T_q^4 \mapsto \{p\}$, and the update happens at program point 4.

The former problem is more intricate. The variables *not* to be updated depend on the solution to T_q^4. Due to loops in the program and self-dependences, the solution to T_q^4 may depend on the variables propagated through program points 3 and 4.

Currently, we have no good solution to this problem.

4.11 Related work

We consider three areas of related work: alias analysis of Fortran and C, the point-to analysis developed by Emami which is the closest related work, and approximation of heap-allocated data structures.

4.11.1 Alias analysis

The literature contains much work on alias analysis of Fortran-like languages. Fortran differs from C in several aspects: dynamic aliases can only be created due to reference parameters, and program's have a purely static call graph.

Banning devised an efficient inter-procedural algorithm for determining the set of aliases of variables, and the side-effects of functions [Banning 1979]. The analysis has two steps. First all trivial aliases are found, and next the alias sets are propagated through the call graph to determine all non-trivial aliases. Cooper and Kennedy improved the complexity of the algorithm by separating the treatment of global variables from reference parameters [Cooper and Kennedy 1989]. Chow has designed an inter-procedural data flow analysis for general single-level pointers [Chow and Rudmik 1982].

Weihl has studied inter-procedural flow analysis in the presence of pointers and procedure variables [Weihl 1980]. The analysis approximates the set of procedures to which a procedure variable may be bound to. Only single-level pointers are treated which is a simpler problem than multi-level pointers, see below. Recently, Mayer and Wolfe have implemented an inter-procedural alias analysis for Fortran based on Cooper and Kennedy's algorithm, and report empirical results [Mayer and Wolfe 1993]. They conclude that the cost of alias analysis is cheap compared to the possible gains. Richardson and Ganapathi have conducted a similar experiment, and conclude that aliases only rarely occur in "realistic" programs [Richardson and Ganapathi 1989]. They also observe that even though inter-procedural analysis theoretically improves the precision of traditional data flow analyses, only a little gain is obtained in actual runtime performance.

Bourdoncle has developed an analysis based on abstract interpretation for computing assertions about scalar variables in a language with nested procedures, aliasing and recursion [Bourdoncle 1990]. The analysis is somewhat complex since the various aspects of interest are computed in parallel, and are not been factored out. Larus *et al.* used a similar machinery to compute inter-procedural alias information [Larus and Hilfinger 1988]. The analysis proceeds by propagating alias information over an extended control-flow graph. Notice that this approach requires the control-flow graph to be statically computable, which is not the case with C. Sagiv *et al.* computes pointer equalities using a similar method [Sagiv and Francez 1990]. Their analysis tracks both universal (must) and existential (may) pointer equalities, and is thus more precise than our analysis. It remains to extend these methods to the full C programming language. Harrison *et al.* use abstract interpretation to analyze program in an intermediate language Mil into which C programs are compiled [Harrison III and Ammarguellat 1992]. Yi has developed a system for automatic generation of program analyses [Yi 1993]. It automatically converts a specification of an abstract interpretation into an implementation.

Landi has developed an inter-procedural alias analysis for a subset of the C language [Landi and Ryder 1992,Landi 1992a]. The algorithm computes flow-sensitive, conditional may-alias information that is used to approximate inter-procedural aliases. The analysis cannot cope with casts and function pointers. Furthermore, its performance is not impressive: 396s to analyze a 3.631 line program is reported.[15] Choi *el al.* have improved on the analysis, and obtained an algorithm that is both more precise and efficient. They use a naming technique for heap-allocated objects similar to the one we have employed. Cytron and Gershbein have developed a similar algorithm for analysis of programs in static single-assignment form [Cytron and Gershbein 1993].

Landi has shown that the problem of finding aliases in a language with more than four levels of pointer indirection, runtime memory allocation and recursive data structures is P-space hard [Landi and Ryder 1991,Landi 1992a]. The proof is by reduction of the set of regular languages, which is known to be P-space complete [Aho *et al.* 1974], to the alias problem [Landi 1992a, Theorem 4.8.1]. Recently it has been shown that intra-procedural may-alias analysis under the same conditions actually not is recursive

[15]To the author knowledge, a new implementation has improved the performance substantially.

[Landi 1992b]. Thus, approximating algorithms are always needed in the case of languages like C.

4.11.2 Points-to analysis

Our initial attempt at pointer analysis was based on abstract interpretation implemented via a (naive) standard iterative fixed-point algorithm. We abandoned this approach since experiments showed that the analysis was far to slow to be feasible. Independently, Emami has developed a *point-to analysis* based on traditional gen-kill data-flow equations, solved via an iterative algorithm [Emami 1993,Emami *et al.* 1993].

Her analysis computes the same kind of information as our analysis, but is more precise: it is flow-sensitive and program-point specific, computes both may and must point-to information, and approximates calls via functions pointers more accurately than our analysis.

The analysis takes as input programs in a language resembling three address code [Aho *et al.* 1986]. For example, a complex statement as `x = a.b[i].c.d[2][j].e` is converted to

```
temp0 = &a.b;
temp1 = &temp0[i];
temp2 = &(*temp1).c.d;
temp3 = &temp2[2][j];
x = (*temp3).e;
```

where the `temp`'s are compile-time introduced variables [Emami 1993, Page 21]. A Simple language may be suitable for machine code generation, but is unacceptably for communication of feedback.

The intra-procedural analysis of statement proceeds by a standard gen-kill approach, where both may and must point-to information is propagated through the control-flow graph. Loops are approximated by a fixed-point algorithm.[16] Heap allocation is approximated very rudely using a single variable "Heap" to represent all heap allocated objects.

We have deliberately chosen to approximate function calls via pointers conservatively, the objective being that more accurate information in the most cases (and definitely for our purpose) is useless. Ghiya and Emami have taken a more advanced approach by using the point-to analysis to perform inter-procedural analysis of calls via pointers. When it has been determined that a function pointer may point to a function f, the call-graph is updated to reflect this, and the (relevant part of the) point-to analysis is repeated [Ghiya 1992].

The inter-procedural analysis is implemented via the program's extended control-flow graph. However, where our technique only increases the number of constraints slightly, Emami's procedure essentially corresponds to copying of the data-flow equations; in practise, the algorithm traverses the (representation) of functions repeatedly. Naturally, this causes the efficiency to degenerate. Unfortunately, we are not aware of any runtime benchmarks, so we can not compare the efficiency of our analysis to Emami's analysis.

[16]Unconditional jumps are removed by a preprocess.

4.11.3 Approximation of data structures

Closely related to analysis of pointers is analysis of heap-allocated data structures. In this chapter we have mainly been concerned with stack-allocated variables, approximating runtime allocated data structures with a 1-limit methods.

Jones and Munchnick have developed a data-flow analysis for inter-procedural analysis of programs with recursive data structures (essentially Lisp S-expressions). The analysis outputs for every program point and variable a regular tree grammar, that includes all the values the variable may assume at runtime. Chase *el al.* improve the analysis by using a more efficient summary technique [Chase *et al.* 1990]. Furthermore, the analysis can discover "true" trees and lists, *i.e.* data structures that contain no aliases between its elements. Larus and Hilfinger have developed a flow analysis that builds an alias graph which illustrates the structure of heap-allocated data [Larus and Hilfinger 1988].

4.12 Further work and conclusion

We have in this chapter developed an inter-procedural point-to analysis for the C programming language, and given a constraint-based implementation. The analysis has been integrated into the *C-Mix* system and proved its usefulness. However, several areas for future work remain to be investigated.

4.12.1 Future work

Practical experiments with the pointer analysis described in this chapter have convincingly demonstrated the feasibility of the analysis, especially with regard to efficiency. The question is whether it is worthwhile to sacrifice some efficiency for the benefit of improved precision. The present analysis approximates as follows:

- flow-insensitive/summary analysis of function bodies,
- arrays are treated as aggregates,
- recursive data structures are collapsed,
- heap-allocated objects are merged according to their birth-place,
- function pointers are not handled in a proper inter-procedurally way.

Consider each in turn.

We considered program-specific pointer analysis in Section 4.10. However, as apparent from the description, the amount of information both during the analysis and in the final result may be too big for practical purposes. For example, in the case of a 1,000 line program with 10 global variables, say, the output will be more than 100,000 state variables (estimating the number of local variables to be 10). Even in the (typical) case of a sparse state description, the total memory usage may easily exceed 1M byte. We identify the main problem to be the following: too much irrelevant information is maintained by the constraint-based analysis. For example, in the state corresponding to the statement '`*p = 1`' the only information of interest is that regarding '`p`'. However, all other state variables are propagated since they may be used at later program points.

We suspect that the extra information contributes only little on realistic programs, but experiments are needed to clarify this. Our belief is that the poor man's approach described in Section 4.2 provides the desired degree of precision, but we have not yet made empirical test that can support this.

Our analysis treats arrays as aggregates. Program using tables of pointers may suffer from this. Dependence analysis developed for parallelizing Fortran compilers has made some progress in this area [Gross and Steenkiste 1990]. The C language is considerably harder to analyze: pointers may be used to reference array elements. We see this as the most promising extension (and the biggest challenge).

The analysis in this chapter merges recursive data structures.[17] In our experience elements in a recursive data structure is used "the same way", but naturally, exceptions may be constructed. Again, practical experiments are needed to evaluate the loss of precision.

Furthermore, the analysis is mainly geared toward analysis of pointers to stack-allocated objects, using a simple notion of (inter-procedural) birth-place to describe heap-allocated objects. Use of birth-time instead of birth-place may be an improvement [Harrison III and Ammarguellat 1992]. In the author's opinion discovery of for instance singly-linked lists, binary trees *etc.* may find substantial use in program transformation and optimization, but we have not investigated inference of such information in detail.

Finally, consider function pointers. The present analysis does not track down inter-procedurally use of function pointers, but uses a sticky treatment. This greatly simplifies the analysis, since otherwise the program's call graph becomes dynamic. The use of static-call graphs is only feasible when *most* of the calls are static. In our experience, function pointers are only rarely used which justifies our coarse approximation, but naturally some programming styles may fail. The approach taken by Ghiya [Ghiya 1992] appears to be expensive, though.

Finally, the relation and benefits of procedure cloning and polymorphic-based analysis should be investigated. The k-limit notions in static-call graphs give a flexible way of adjusting the precision with respect to recursive calls. Polymorphic analyses are less flexible but seem to handle program with dynamic call graphs more easily.

4.12.2 Conclusion

We have reported on an inter-procedural, flow-insensitive point-to analysis for the entire C programming language. The analysis is founded on constraint-based program analysis, which allows a clean separation between specification and implementation. We have devised a technique for inter-procedural analysis which prevents copying of constraints. Furthermore we have given an efficient algorithm.

[17]This happens as a side-effect of the program representation, but the k-limit can easily be increased.

Chapter 5

Binding-Time Analysis

We develop an efficient binding-time analysis for the ANSI C programming language. The aim of binding-time analysis is to classify constructs (variables, expressions, statements, functions, ...) as either compile-time or run-time, given an initial division of the input. Evidently, a division where all variables are classified as run-time is correct but of no value. We seek a most static annotation that does not break the congruence principle: a construct that depends on a run-time value must be classified as run-time. More precisely, the analysis computes a polyvariant, program-point insensitive division.

Explicit separation of binding-times has turned out to be crucial for successful self-application of partial evaluators. It also seems an important stepping stone for specialization of imperative language featuring pointers, user-defined structs and side-effects. Binding-time analysis is the driving part of a generating-extension transformation.

Given a program, the analysis annotates all type specifiers with a binding time. For example, an integer pointer may be classified as a "static pointer to a dynamic object". We present various extensions that may enhance the result of program specialization.

The analysis is specified by the means of non-standard type system. Given a program where run-time constructs are marked, the rules check the consistency of the annotation. A program satisfying the type systems is well-annotated.

The type systems are then formulated in a constraint-based framework for binding-time inference. The constraints capture the dependencies between an expression and its subexpressions. The constraint system consists of constraints over binding-time attributed types. It is shown that a solution to a constraint system yields a well-annotated program. An extension is given that allows context-sensitive analysis of functions, based on the program's static-call graph.

An efficient constraint-solver is developed. The algorithm exhibits an amortized run-time complexity which is almost-linear, and in practice it is extremely fast. We have implemented and integrated the analysis into the *C-Mix* partial evaluator.

Part of this chapter is based on previous work on binding-time analysis for self-applicable C partial evaluation [Andersen 1993a,Andersen 1993b]. It has been extended to cover the full ANSI C language; the constraint formulation has been simplified, and a new and faster algorithm developed. Furthermore, the context-sensitive, or polyvariant, inference has been added. Part of this chapter is based on the paper [Andersen 1993a].

5.1 Introduction

A binding-time analysis takes a program and an initial division of the input into static (compile-time) and dynamic (run-time), and computes a division classifying all expressions as being either static or dynamic. An expression that depends on a dynamic value must be classified dynamic. This is the so-called *congruence principle.*

Explicit separation of binding times is useful in a number of applications. In constant folding, static expressions depend solely on available values, and can thus be evaluated at compile-time. In partial evaluation, static constructs can be evaluated at specialization time while residual code must be generated for dynamic expressions. In a generating-extension transformation, dynamic constructs are changed to code generating expressions.

5.1.1 The use of binding times

Binding-time analysis was introduced into partial evaluation as a means to obtain efficient self-application of specializers. Assume that '`int`' is an interpreter and consider self-application of partial evaluator '`mix`'.

$$[\![\mathtt{mix}_1]\!](\mathtt{mix}_2, \mathtt{int}) \Rightarrow \mathtt{comp}$$

During self-application, the second '`mix`' ('$\mathtt{mix}_2$') cannot "see" the "natural" binding times in '`int`': that the program is available when the input is delivered. Thus, the resulting compiler must take into account that, at compile-time, only the input is given. This is excessively general since the program *will* be available at compile time. The problem is solved by annotating '`int`' with binding times that inform '`mix`' that the program is truly compile-time [Jones *et al.* 1993, Chapter 7]

Explicit binding-time separation seems also important for successful specialization of imperative languages featuring pointers, user-defined structs, run-time memory allocations and side-effects. Consider for example specialization of the following function.

```
int foo(void)
{
   int x, *p;
   p = &x;
   return bar(p);
}
```

Must the address operator application be suspended? An on-line partial evaluator cannot decide this since it in general depends on '`bar()`'s usage of its parameter. A binding-time analysis collects global information to determine the binding time of '`p`', and thus whether the application must be suspended.

The first binding-time analyses treated small first-order Lisp-like programs, and were implemented via abstract interpretation over the domain $\{S \sqsubset D\}$ [Jones *et al.* 1989]. Data structures were considered aggregates such that, for example, an alist with a static key but a dynamic value would be approximated by 'dynamic'. Various analyses coping with partially-static data structures and higher-order languages have later been developed [Bondorf 1990,Consel 1993a,Launchbury 1990,Mogensen 1989].

Recently, binding-time analyses based on non-standard type inference have attracted much attention [Birkedal and Welinder 1993,Bondorf and Jørgensen 1993,Gomard 1990, Henglein 1991,Nielson and Nielson 1988]. They capture in a natural way partially-static data structures as well as the higher-order aspect of functional languages, and accommodate efficient implementations [Henglein 1991].

5.1.2 Efficient binding-time analysis

Even though many program analyses specified by the means of type systems can be implemented by (slightly) modified versions of the standard type-inference algorithm W, this approach seems to give too in-efficient analyses [Andersen and Mossin 1990,Gomard 1990, Nielson and Nielson 1988].

Based on the ideas behind semi-unification, Henglein reformulated the problem and gave an efficient *constraint-set solving* algorithm for an untyped lambda calculus with constants and a fixed-point operator [Henglein 1991]. The analysis exhibits an amortized run-time complexity that is almost-linear in the size of the input program.

The overall idea is to capture dependencies between an expression and its subexpressions via *constraints*. For example, in the case of an expression

$$e \equiv \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,$$

the following constraints could be generated:

$$\{T_{e_2} \preceq T_e, T_{e_3} \preceq T_e, T_{e_1} \,\rhd T_e\}$$

where T_e is a *binding-time variable* assigned to expression e.

The first two constraints say that the binding time of the expression e is greater then or equal to the binding time of the returned values (where $S \prec D$). The last constraint captures that if the test expression is dynamic, then so must the whole expression. A solution to a constraint system is a substitution of S or D for type variables such that all constraints are satisfied.

The analysis proceeds in three phases. First, the constraint set is *collected* during a syntax-directed traversal of the program. Next, the constraints are *normalized* by exhaustive application of a set of rewrite rules. Finally the constraint set is *solved*, which turns out to be trivial for normalized systems.

5.1.3 Related work

Since we originally formulated a binding-time analysis for C [Andersen 1991], and implemented it [Andersen 1993a], Bondorf and Jørgensen have developed a similar constraint-set based analysis for the Similix partial evaluator [Bondorf and Jørgensen 1993]. Their analysis treats a higher-order subset of the Scheme programming language and supports partially static data structures. The semantics of the Similix specializer is somewhat different from the semantics of our specializer, which in some respects simplifies their analysis. For example, in Similix, structured values passed to an external function (primitive operator) are suspended. This is too conservative in the case of C.

Birkedal and Welinder have recently formulated and implemented a binding-time analysis for the Standard ML language [Birkedal and Welinder 1993].

5.1.4 The present work

This chapter contributes with three main parts. We specify *well-separation* of binding time annotations, hereby giving a correctness criteria for the analysis. The definition of well-annotatedness naturally depends on the *use* of the annotations. For example, the generating-extension transformation of Chapter 3 imposes several restrictions upon binding times, *e.g.* that side-effects under dynamic control must be suspended.

Next we present a *monovariant* binding-time analysis, and extend it to a context-sensitive *polyvariant* analysis. Normally, context-sensitive constraint-based analysis is achieved by explicit copying of constraints; we give an approach where constraints over vectors are generated. This way, the number of constraints does not grow exponentially.

Example 5.1 Consider the following C program.

```
int pows(int x, int y)        int pow(int base, int x)
{                             {
   pow(x,y);                     int p = 1;
   pow(y,x);                     while (base--) p *= x;
}                                return p;
                               }
```

Suppose that ‘x’ is static and ‘y’ is dynamic. A context-insensitive analysis will merge the two calls to ‘pow(), and henceforth classify as dynamic both parameters. A context-sensitive analysis makes two annotations of the function: one where ‘base’ is static and ‘x’ is dynamic, and vice versa. **End of Example**

Finally, we develop an *efficient inference algorithm* that runs in almost-linear time. The algorithm is based on efficient find-union data structures.

We assume the existence of *pointer information.* That is, for every variable p of pointer type, an approximation of the set of objects p may point to at run-time. Furthermore, we assume that assignments are annotated as *side-effecting* or *conditionally side-effecting* if it may assign a non-local object, or assign a non-local object under conditional control, respectively. These annotations can be computed by the pointer analysis of Chapter 4 and the side-effect analysis of Chapter 6.

5.1.5 Overview of the chapter

The chapter is organized as follows. Section 5.2 discusses some intriguing constructs in the C language. Section 5.3 defines *well-annotated* programs. Section 5.4 develops a *constraint-based* formulation. Section 5.5 presets an efficient *normalization* algorithm. In Section 5.6 we extend the analysis into a *polyvariant* analysis. Section 5.7 contains some examples, Section 5.8 describes related work, and Section 5.10 holds the conclusion and list topics for further work.

5.2 Separating binding times

A classification of variables[1] into static and dynamic is called a *division*. A necessary requirement for a binding-time analysis is the so-called *congruence principle*: a static variable must not depend on a dynamic variable. Furthermore, it must fulfill the requirements imposed by the generating-extension transformation, as listed in Chapter 3. We briefly review these when needed. In this section we consider some intricate aspects of the C language.

5.2.1 Externally defined identifiers

A C program normally consists of a set of translation units. Identifiers defined in other modules than the one being analyzed are *external*. An externally defined identifier is brought into scope via an '`extern`' declaration. Global identifiers are by default "exported" to other modules unless explicitly declared to be local by the means of the '`static`' storage specifier.

Example 5.2 The program below consists of two files. File 2 defines the function '`pow()`' which is used in file 1.

```
/* File 1 */                          /* File 2 */
extern int errno;
extern double my_pow(double,double);   int errrno;
int main(void);                        double pow(double b,double x)
{                                      {
   double p = pow(2,3);                   if (x > 0.0)
   if (!errno)                               return exp(x * log(b));
      printf("Pow = %f\n", p);            errno = EDOM;
   return errno;                          return 0.0;
}                                      }
```

Looking isolated at file 2, it cannot be determined whether the global variable '`errno`' is used (and even assigned) by other modules. Further, the calls to '`pow()`' are unknown. When looking at file 1, the values assigned to '`errno`' are unknown. **End of Example**

This have two consequences: for the analysis to be safe,

- *all* global variables must be classified dynamic, and
- *all* functions must be annotated dynamic. This restriction can be alleviated by copying all functions, and annotating the copy completely dynamic.

Naturally, this is excessively strict, and it obstructs possible gains from specialization. Inter-modular information (or user guidance) is needed to make less conservative assumptions. This is the subject of Chapter 7 that considers *separate* program analysis.

In this chapter we adopt the following assumptions.

[1] C allows complex data structures so 'variables' should be replaced by 'objects'.

Constraint 5.1

- We analyze *programs* that consist of one or more translation units.[2]
- Only calls to library functions are 'extern'.[3]
- Other modules do not refer to global identifiers. This means that global identifiers are effectively considered 'static'.

Chapter 7 develops an analysis that eliminates these requirements.

5.2.2 Pure functions

Calls to externally defined functions must be suspended since, in general, the operational behavior of such functions are unknown. For example, a function may report errors in a global variable, as is the case for most functions defined in the `<math.h>` library.

Example 5.3 Suppose an external function '`count()`' counts the number of times a particular part of a program is invoked.

```
/* File 1 */                      /* File 2 */
extern void count(void);
int main(void)                     void count(void)
{                                  {
   ...                                static int n = 0;
   count();                           n++;
   ...                                return;
   return 0;                       }
}
```

Evaluation of a call '`count()`' during specialization is dubious due to the side-effects on variable '`n`'. More dramatic examples include functions that output data or abort program execution. **End of Example**

This means that calls such as '`sin(M_PI)`' will be annotated dynamic, and thus not replaced by a constant during specialization, as would be expected. To ameliorate the situation, an externally function can be specified as *pure*.

Definition 5.1 *A function is* pure *if it commits no side-effects, i.e. assignments to non-local objects.* □

In this chapter we assume that function are pre-annotated by '`pure`' specifiers. The annotations can be derived by the side-effect analysis of Chapter 6, or be given as user specifications.

[2]A 'program' does not necessary have to be complete.

[3]A 'library function' is here be interpreted as a function defined independently of the program.

5.2.3 Function specialization

A function can be handled in essentially three different ways by a partial evaluator. It can be specialized and possibly shared between several residual calls; it can be unconditionally specialized such that no sharing is introduced, or it can be unfolded. The treatment of a function may influence the binding times. For example, the parameters of an unfolded function may be assigned partially-static binding times, whereas other functions must have either completely static or completely dynamic parameters.

In this chapter we will for ease of presentation assume that all functions are *specialized* and possibly shared. We describe the modifications needed to support analysis of unfoldable functions in Section 5.4.8.

5.2.4 Unions and common initial members

The Standard specifies that common initial members in members of struct type in a union shall be truly shared, cf. Section 2.4. This implies that these fields must be given the same binding time.

For simplicity we ignore the problem in the later exposition. The rule is easy to implement but ugly to describe. The idea is to generate constraints that relate the common members.

5.2.5 Pointers and side-effects

A side-effect is either due to an assignment where the left hand side expression evaluates to the address of a non-local object, or run-time memory allocation, which side-effects the heap.

We will assume that assignments are annotated with *side-effect* information. The annotation † on an assignment $e_1 =^{\dagger} e_2$ means that the expression (may) side-effect. The annotation ‡ on an assignment $e_1 =^{\ddagger} e_2$ means that the expression (may) side-effect under conditional control. See Chapter 6 for definition and computation of side-effects annotations.

The generating-extension transformation requires that side-effects under dynamic control shall be suspended. Initially, we will suspend side-effects under conditional control; a later extension develops suspension of side-effects under dynamic control only (Section 5.4.8).

5.2.6 Run-time memory allocation

Run-time memory allocation by the means of '`malloc()`' (or one of its derived forms) shall be suspended to run-time. Only allocation via the 'mix' function '`alloc()`' may be performed at specialization time.

We assume that all '`alloc`^{l}`()`' calls are labeled uniquely. The label l is employed to indicate the binding time of objects allocated by the l'th call-site. In the following we often implicitly assume definitions of the form '`struct S *l`' for all alloc calls '`alloc`^{l}`(S)`'.

Heap-allocated objects are anonymous global objects. Recall that in order to compare function calls to detect sharing of residual functions, call signatures must be compared. This may be time-consuming, and it may thus be desirable to prevent specialization with respect to heap-allocated data.

More broadly, we assume a specifier '`residual`' that unconditionally shall suspend a variable. For example, when a programmer defines '`residual int *p`' the pointer '`p`' shall be classified dynamic.

5.2.7 Implementation-defined behaviour

A program can be *conforming* or *strictly conforming* to the ANSI C Standard [ISO 1990]. A strictly conforming program is not allowed to depend on undefined, unspecified or implementation-defined behavior. A conforming program may rely on implementation-defined behavior. An example of a non-strictly conforming feature is cast of integral values to pointers. Implementation-defined features shall be suspended to run-time.

Example 5.4 The cast in '`(int *)0x2d0c`' must be annotated dynamic since cast of an integral value to a pointer is implementation-defined.

The result of the '`sizeof()`' is implementation-defined. Thus, the application shall be annotated dynamic. **End of Example**

A special class of implementation-defined operations is *casts*. Cast between base types presents no problem: if the value is static the cast can be performed during specialization; otherwise it must be annotated dynamic. Below we will often ignore casts between base types, and only consider casts involving a pointer.

5.2.8 Pointers: casts and arithmetic

The C language supports cast between pointers and integral values, and pointer arithmetic. For example, the cast '`(size_t)p`' casts a pointer '`p`' to a "`size_t`" value, and '`p + 2`' increases the pointer by 2.

Consider first cast of pointers which can be divided into four group [ISO 1990, Paragraph 6.3.4].

1. Cast of a pointer to an integral values. The result is implementation-defined.
2. Cast of an integral value to a pointer. The result is implementation-defined.
3. Cast of a pointer type to another pointer type with less alignment requirement and back again. The result shall compare equal with the original pointer.
4. A function pointer can be cast to another function pointer and back again, and the result shall compare equal with the original value. If a converted pointer is used to call a function with a incompatible type, the result is undefined.

In the two first cases, the cast shall be annotated dynamic. In case three and four, the cast can be annotated static provided the pointers are static.[4]

Consider now pointer arithmetic. An integer value may be added to a pointer, and two pointers may be subtracted. In both cases, for the operation to be static, both operands must be static.

Example 5.5 Consider an expression '`p + 1`' which adds one to a pointer. Suppose that '`p`' is a "static" pointer to a "dynamic" object. Normally, an operator requires its arguments to be fully static. In this case, the indirection of '`p`' is not needed to carry out the addition, and henceforth is should be classified static despite '`p`' being partially-static. We consider this extension in Example 5.14. **End of Example**

5.3 Specifying binding times

Suppose a program is given where all functions, statements and expressions are marked static (compile time) or dynamic (run time). This section defines a set of rules that checks whether the annotations are placed consistently. A program where the binding time separation fulfills the congruence principle and the additional requirements imposed by the generating-extension transformation is *well-annotated*.

The analysis developed in this chapter computes an annotation of expressions. In Section 5.4.7 we describe how a well-annotated program can be derived from an expression annotation, and furthermore, how a well-annotated program can be derived from a variable division.

5.3.1 Binding-time types

The *type* of an expression describes the value the expression evaluates to. The *binding time* of an expression describes *when* the expression can be evaluate to its value. Thus, there is an intimate relationship between types and binding times.

A binding time B can be static or dynamic:

$$B ::= S \mid D \mid \beta$$

where β is a *binding time variable* ranging over S and D. We use B to range over binding times. To model the binding time of a value, we extend static types to include binding time information. For example, a static pointer '`p`' to a dynamic integer value is denoted by the *binding-time type* $\mathtt{p} : \left\langle {* \atop S} \right\rangle \left\langle {\mathtt{int} \atop D} \right\rangle$.

Definition 5.2 *The syntax of a binding time type BT is defined inductively by :*

$$\begin{array}{rcll}
BT & ::= & \left\langle {\tau_{\mathtt{b}} \atop B} \right\rangle & \textit{Base type} \\
 & \mid & \left\langle {\tau_{\mathtt{s}} \atop B} \right\rangle & \textit{Struct type} \\
 & \mid & \left\langle {* \atop B} \right\rangle BT & \textit{Pointer type} \\
 & \mid & \left\langle {[\mathtt{n}] \atop B} \right\rangle BT & \textit{Array type} \\
 & \mid & \left\langle {(\mathtt{BT}^*) \atop B} \right\rangle BT & \textit{Function type}
\end{array}$$

[4]The notion of static pointers are defined in the next section.

where τ_b range over base type specifiers, and τ_s range over struct/union/enumerator type specifiers. If $T = \left\langle {\tau \atop B} \right\rangle T_1$, then $T\#b$ denotes the binding time of T, i.e. $T\#b = B$. □

A binding-time type (bt-type) is an attributed type. We use T to range over both normal types[5] and bt-types when no confusion is likely to occur. We will occasionally use BT to range over bt-types when needed.

Example 5.6 Let the definitions '`int x, a[10], *p`' be given and consider the bt-types:

$$\begin{array}{lcl}
\texttt{int x} & : & \left\langle {\texttt{int} \atop S} \right\rangle \\
\texttt{int a}[10] & : & \left\langle {[10] \atop D} \right\rangle \left\langle {\texttt{int} \atop D} \right\rangle \\
\texttt{int} \ *\texttt{p} & : & \left\langle {* \atop S} \right\rangle \left\langle {\texttt{int} \atop D} \right\rangle
\end{array}$$

The variable '`x`' is static. The array '`a`' is completely dynamic. The pointer '`p`' is a static pointer to a dynamic object. If $T = \left\langle {* \atop S} \right\rangle \left\langle {\texttt{int} \atop D} \right\rangle$, then $T\#b = S$. **End of Example**

A pointer is a variable containing an address or the constant '`NULL`'. If the address is definitely known at specialization time, the pointer can be classified static. Otherwise it must be classified dynamic. Static pointers can be dereferenced during specialization; dynamic pointers cannot. Naturally it makes no sense to classify a pointer "dynamic to a static object", since the object a dynamic pointer points to must exist in the residual program. This is captured by the following definition of *well-formed types.*

Definition 5.3 *A bt-type T is* well-formed *if it satisfies the following requirements.*

1. *If $T = \left\langle {\tau_1 \atop B_1} \right\rangle \ldots \left\langle {\tau_n \atop B_n} \right\rangle$ and there exists an i s.t. $B_i = D$, then for all $j > i$: $B_j = D$.*
2. *If $\left\langle {(\texttt{T}_1,\ldots,\texttt{T}_n) \atop B} \right\rangle$ in T, then if there exists an i s.t. $T_i\#b = D$ then $B = D$.*

□

The first condition stipulates that if a variable of pointer type is dynamic, then so must the "dereferenced" type be. The second part states that if a "parameter" in a function type is dynamic, the binding time of the specifier must be dynamic. Intuitively, if the binding time of a function type specifier is dynamic, the function takes a dynamic argument.

Example 5.7 The type $\left\langle {* \atop D} \right\rangle \left\langle {\texttt{int} \atop S} \right\rangle$ is not well-formed. The type $\left\langle {* \atop S} \right\rangle \left\langle {() \atop D} \right\rangle \left\langle {\texttt{int} \atop D} \right\rangle$ is well-formed. **End of Example**

[5]In the following we often write 'static type' meaning the static program type — not a bt-type where all binding times are static.

5.3.2 Binding time classifications of objects

Arrays are treated as aggregates, and thus all entries of an array are assigned the same binding times. For example, the bt-type $\left\langle\substack{\texttt{[n]}\\ S}\right\rangle\left\langle\substack{\texttt{int}\\ D}\right\rangle$ specifies the type of a static array with dynamic content. An array is said to be static if it can be "indexed" during specialization. In this example, it would yield a dynamic object.

Example 5.8 Suppose a pointer '`p`' is classified by `p` : $\left\langle\substack{*\\ S}\right\rangle\left\langle\substack{\texttt{int}\\ S}\right\rangle$, and consider the expression '`p + 1`'. For the expression to make sense, '`p`' must point into an array, '`a`', say. Observe that pointer arithmetic cannot "change" the bt-type of a pointer, *e.g.* '`p+1`' cannot point to a dynamic object, since pointer arithmetic is not allowed to move a pointer outside an array, (*i.e.* '`a`'), and arrays are treated as aggregates. **End of Example**

Structs are classified static or dynamic depending on whether they are split. Static structs are split into individual variables (or eliminated) during specialization. Given the bt-type of a variable of struct type: $\left\langle\substack{\texttt{struct S}\\ B}\right\rangle$, the binding time B indicates whether the struct is split. If B equals D, it is not split, and all members of S shall be dynamic. Otherwise the struct is split.

5.3.3 The lift relation

Suppose that '`x`' is an integer variable, and consider the assignment '`x = c`'. Even though '`x`' is dynamic, '`c`' is allowed to be static — it can be *lifted* to run-time. Operationally speaking, a run-time constant can be built from the value of c at specialization time.

This is not the case for objects of struct type. The 'obvious' solution would be to introduce 'constructor' functions, but this may lead to code duplication and introduces an overhead. Objects of struct type cannot be lifted. Similarly holds for pointers, since lifting of addresses introduces an implementation dependency. The notion of values that can be lifted is captured by the relation $<$ on bt-types defined as follows.

Definition 5.4 *Define the relation* '$<$: BType $\times$ BType' *by* $\left\langle\substack{\tau_b\\ S}\right\rangle < \left\langle\substack{\tau_b\\ D}\right\rangle$, *and* $T_1 \leq T_2$ *iff* $T_1 < T_2$ *or* $T_1 = T_2$.[6] □

The definition says that only a static value of base type can be lifted to a dynamic value (of same base type).

Observe that it can be determined on the basis of static program types where lift (possibly) can occur. For example, in an assignment 'e_1 `=` e_2' where e_1 is of struct type, no lift operator can possibly be applied, since objects of struct type cannot be lifted.

Example 5.9 Where can a base type value be lifted? It can happen at one of the following places: the index of an array indexing, arguments to operator application, arguments to function calls, and the right hand side expressions of assignments. Further, the subexpression of a comma expression may be lifted, if the other is dynamic. Finally, the value returned by a function may be lifted. **End of Example**

[6] We will later allow lift of string constants.

5.3.4 Divisions and type environment

A *binding-time environment* '$\widetilde{\mathcal{E}}$: Id $\rightarrow$ BType' is a map from identifiers to bt-types. Given a type T and a bt-type BT, BT *suits* T if they agree on the static program type. An environment *suits* a set of definitions $x : T_x$ if it is defined for all x and $\widetilde{\mathcal{E}}(x)$ suits T_x.

A *type environment* '$\widetilde{\mathcal{TE}}$: TName $\rightarrow$ BType $\times$ (Id $\rightarrow$ BType)' is a map from type names to bt-types and binding-time environments. Intuitively, if S is a struct type name, and $\widetilde{\mathcal{TE}}(S) = \left\langle \left\langle {\texttt{structS} \atop B} \right\rangle, \widetilde{\mathcal{E}}' \right\rangle$, then B indicates whether the struct is static (*i.e.* can be split), and $\widetilde{\mathcal{E}}'$ represents the bt-types of the members (*e.g.* $\widetilde{\mathcal{TE}}(S) \downarrow 2(x)$ is the binding time of a member '`x`').

Example 5.10 Define '`struct S { int x; struct S *next; }`'. The map

$$\widetilde{\mathcal{TE}} = \left[S \mapsto \left\langle \left\langle {\texttt{struct S} \atop S} \right\rangle, \left[x \mapsto \left\langle {\texttt{int} \atop D} \right\rangle, next \mapsto \left\langle {* \atop S} \right\rangle \left\langle {\texttt{struct S} \atop S} \right\rangle \right] \right\rangle \right]$$

describes that member '`x`' is dynamic, '`next`' is a static pointer, and '`struct S`' can be split.

For ease of notation, we omit product projections when they are clear from the context. For example, we write $\widetilde{\mathcal{TE}}(S)$ for the binding time of S, and $\widetilde{\mathcal{TE}}(S)(x)$ for the binding time of member x. **End of Example**

To be consistent, the binding time of a struct specifier of a variable must agree with the binding time recorded by a type environment.

Definition 5.5 *Let $\widetilde{\mathcal{E}}$ be a binding time environment and $\widetilde{\mathcal{TE}}$ a type environment. Then $\widetilde{\mathcal{E}}$ is said to* agree *with $\widetilde{\mathcal{TE}}$ if for every $x \in dom(\widetilde{\mathcal{E}})$, if $\widetilde{\mathcal{E}}(x)$ contains a specifier $\left\langle {\texttt{struct S} \atop B} \right\rangle$, then $\widetilde{\mathcal{TE}}(S) \downarrow 1 = \left\langle {\texttt{struct S} \atop B} \right\rangle$, and similarly for '`union`'.* □

An *operator type assignment* is a map 'O : Op $\rightarrow$ BType' assigning bt-types to operators, where all binding times are variables. Let 'stat : BType $\rightarrow$ BType' be the function that returns a copy of a bt-type where all binding times are S. Let 'dyn : BType $\rightarrow$ BType' be defined similarly.

Example 5.11 Consider the "integer plus" $+_{int,int}$ operator, and the "pointer and integer" $+_{ptr,int}$ operator. Then $O(+_{int,int}) = \left\langle {(\left\langle {\texttt{int} \atop \beta_1} \right\rangle, \left\langle {\texttt{int} \atop \beta_2} \right\rangle) \atop B} \right\rangle \left\langle {\texttt{int} \atop \beta} \right\rangle$, and $O(+_{ptr,int}) = \left\langle {(\left\langle {* \atop \beta_1} \right\rangle \left\langle {\texttt{int} \atop \beta_2} \right\rangle, \left\langle {\texttt{int} \atop \beta_3} \right\rangle) \atop B} \right\rangle \left\langle {* \atop \beta_4} \right\rangle \left\langle {\texttt{int} \atop \beta_5} \right\rangle$.

We have 'dyn$(\left\langle {* \atop S} \right\rangle \left\langle {\texttt{int} \atop D} \right\rangle) = \left\langle {* \atop D} \right\rangle \left\langle {\texttt{int} \atop D} \right\rangle$'. **End of Example**

Recall that overloading of operators is assumed to be resolved during parsing.

5.3.5 Two-level binding-time annotation

We adopt the *two-level language* framework for specifying binding times in programs [Nielson and Nielson 1992a,Gomard and Jones 1991b]. An underlined construct is dynamic while other constructs are static. For example '$e_1\texttt{[}e_2\texttt{]}$' denotes a static array indexing, and '$\underline{e_1\texttt{[}e_2\texttt{]}}$' denotes a dynamic index.[7]

Example 5.12 The well-known '`pow()`' program can be annotated as follows, where '`base`' is static and '`x`' is dynamic.

$$\begin{array}{l}
\underline{\texttt{int pow}}\texttt{(int base, }\underline{\texttt{int x}}\texttt{)} \\
\{ \\
\quad \underline{\texttt{int p = 1}}\texttt{;} \\
\quad \texttt{for (; base--;) }\underline{\texttt{p *= x}}\texttt{;} \\
\quad \underline{\texttt{return p}}\texttt{;} \\
\}
\end{array}$$

It is easy to see that the binding time annotations are "consistent" with the initial binding time division. **End of Example**

We omit a formal specification of two-level C. A two-level version of a subset of C has previously been defined [Andersen 1993b].

5.3.6 Well-annotated definitions

Let $d \equiv x : T_x$ be an annotated declaration. It gives rise to a bt-type BT_x that suits T_x. The relation

$$\vdash^{decl}: \text{Decl} \times \text{BType}$$

defined in Figure 44 captures this.

Definition 5.6 *Let d be an annotated declaration, and $\widetilde{\mathcal{TE}}$ a type environment defined for all type names in d. Then the bt-type BT agree with the type of d if*

$$\widetilde{\mathcal{TE}} \vdash^{decl} d : BT$$

where $\vdash^{decl}$ is defined in Figure 44. □

The definition of $\vdash^{decl}$ uses the relation $\vdash^{type}$: Type × BType which also is defined by Figure 44.

In the case of a "pure" declaration, the binding time of the type must be static. Otherwise it must be dynamic. In the case of definitions, an underlined definition must posses a dynamic type. Definitions specified '`residual`' must be dynamic.

The rule for function types uses an auxiliary predicate 'statdyn'. It is satisfied when the binding time type is completely static or completely dynamic, respectively.[8] The use of 'statdyn' makes sure that a function does not accept partially static arguments.

The 'statdyn' predicate could have been specified by an additional set of inference rules for types, but is omitted due to lack of space.

[7]Even though it would be more correct to underline the expression constructor, we underline for the sake of readability the whole expression.

[8]In the case where a type specifier is a struct, the members must be checked as well.

[decl]	$\dfrac{\widetilde{\mathcal{TE}} \vdash^{type} T : BT, BT\#b = S}{\widetilde{\mathcal{TE}} \vdash^{decl} \texttt{pure extern}\ x : T : BT}$	$\dfrac{\widetilde{\mathcal{TE}} \vdash^{type} T : BT, BT\#b = D}{\widetilde{\mathcal{TE}} \vdash^{type} \underline{\texttt{extern}\ x : T} : BT}$
[def]	$\dfrac{\widetilde{\mathcal{TE}} \vdash^{type} T : BT, BT\#b = S}{\widetilde{\mathcal{TE}} \vdash^{decl} x : T\ \ : BT}$	$\dfrac{\widetilde{\mathcal{TE}} \vdash^{type} T : BT, BT\#b = D}{\widetilde{\mathcal{TE}} \vdash^{decl} \underline{x : T} : BT}$
[res]		$\dfrac{\widetilde{\mathcal{TE}} \vdash^{type} T : BT, BT\#b = D}{\widetilde{\mathcal{TE}} \vdash^{decl} \underline{\texttt{residual}\ x : T} : BT}$
[base]	$\widetilde{\mathcal{TE}} \vdash^{type} \langle \tau_b \rangle : \left\langle \genfrac{}{}{0pt}{}{\tau_{\texttt{b}}}{S} \right\rangle$	$\widetilde{\mathcal{TE}} \vdash^{type} \underline{\langle \tau_b \rangle} : \left\langle \genfrac{}{}{0pt}{}{\tau_{\texttt{b}}}{D} \right\rangle$
[struct]	$\dfrac{\widetilde{\mathcal{TE}}(S) = \left\langle \genfrac{}{}{0pt}{}{\texttt{struct S}}{S} \right\rangle}{\widetilde{\mathcal{TE}} \vdash^{type} \langle \text{struct S} \rangle : \left\langle \genfrac{}{}{0pt}{}{\texttt{struct S}}{S} \right\rangle}$	$\dfrac{\widetilde{\mathcal{TE}}(S) = \left\langle \genfrac{}{}{0pt}{}{\texttt{struct S}}{D} \right\rangle}{\widetilde{\mathcal{TE}} \vdash^{type} \underline{\langle \text{struct S} \rangle} : \left\langle \genfrac{}{}{0pt}{}{\texttt{struct S}}{D} \right\rangle}$
[ptr]	$\dfrac{\widetilde{\mathcal{TE}} \vdash^{type} T : BT}{\widetilde{\mathcal{TE}} \vdash^{type} \langle * \rangle T : \left\langle \genfrac{}{}{0pt}{}{*}{S} \right\rangle BT}$	$\dfrac{\widetilde{\mathcal{TE}} \vdash^{type} T : BT, BT\#b = D}{\widetilde{\mathcal{TE}} \vdash^{type} \underline{\langle * \rangle T} : \left\langle \genfrac{}{}{0pt}{}{*}{D} \right\rangle BT}$
[array]	$\dfrac{\widetilde{\mathcal{TE}} \vdash^{type} T : BT}{\widetilde{\mathcal{TE}} \vdash^{type} \langle [n] \rangle T : \left\langle \genfrac{}{}{0pt}{}{\texttt{[n]}}{S} \right\rangle BT}$	$\dfrac{\widetilde{\mathcal{TE}} \vdash^{type} T : BT, BT\#b = D}{\widetilde{\mathcal{TE}} \vdash^{type} \underline{\langle [n] \rangle T} : \left\langle \genfrac{}{}{0pt}{}{\texttt{[n]}}{D} \right\rangle BT}$
[fun]	$\dfrac{\begin{matrix}\widetilde{\mathcal{TE}} \vdash^{decl} d_i : BT_i, \text{statdyn}(BT_i) \\ \widetilde{\mathcal{TE}} \vdash^{type} T : BT, BT_i\#b = S\end{matrix}}{\widetilde{\mathcal{TE}} \vdash^{type} \langle (d_i) \rangle T : \left\langle \genfrac{}{}{0pt}{}{(BT_i)}{S} \right\rangle BT}$	$\dfrac{\begin{matrix}\widetilde{\mathcal{TE}} \vdash^{decl} d_i : BT_i, \text{statdyn}(BT_i) \\ \widetilde{\mathcal{TE}} \vdash^{type} T : BT, BT\#b = D\end{matrix}}{\widetilde{\mathcal{TE}} \vdash^{type} \underline{\langle (d_i) \rangle T} : \left\langle \genfrac{}{}{0pt}{}{(BT_i)}{D} \right\rangle BT}$

Figure 44: Binding time inference rules for declarations

Lemma 5.1 *If d is a declaration and T a type such that $\widetilde{\mathcal{TE}} \vdash^{decl} d : T$, then T is well-formed.*

Proof Suppose that $\widetilde{\mathcal{TE}} \vdash^{decl} d : T$, and consider the two condition in Definition 5.3. The first is satisfied due to the condition $BT\#b = D$ in the dynamic "versions" of the rules. The second condition is fulfilled due to the condition $BT_i\#b = S$ in the static version of the rule for function type specifiers. □

Suppose that D is a type definition and $\widetilde{\mathcal{TE}}$ is a type environment. The agreement between a type definition and an annotated definition is captured by

$\vdash^{tdef}$: TDef

defined in Figure 45.

Definition 5.7 *Let D be an annotated type definition. The type environment $\widetilde{\mathcal{TE}}$ agrees with the annotation of D if*

$\widetilde{\mathcal{TE}} \vdash^{tdef} D : \bullet$

where $\vdash^{tdef}$ is defined in Figure 45. □

Lemma 5.2 *Let $\widetilde{\mathcal{TE}}$ be a type environment, and define $\widetilde{\mathcal{E}} = [x \mapsto BT]$ for declaration $x : T$ where $\widetilde{\mathcal{TE}} \vdash^{decl} x : T\ \ :\ BT$. Then $\widetilde{\mathcal{E}}$* agrees *with $\widetilde{\mathcal{TE}}$.*

Proof Obvious. □

$$\text{[struct]}\quad \frac{\begin{array}{l}\widetilde{\mathcal{TE}}(S) = \left\langle \begin{array}{c}\texttt{struct S}\\ S\end{array}\right\rangle \\ \widetilde{\mathcal{TE}} \vdash^{type} T_x : \widetilde{\mathcal{TE}}(S)(x)\end{array}}{\widetilde{\mathcal{TE}} \vdash^{tdef} \texttt{struct S \{ } x : T_x \texttt{ \}} : \bullet} \qquad \frac{\begin{array}{l}\widetilde{\mathcal{TE}}(S) = \left\langle \begin{array}{c}\texttt{struct S}\\ D\end{array}\right\rangle \\ \widetilde{\mathcal{TE}} \vdash^{type} T_x : \widetilde{\mathcal{TE}}(S)(x), T_x\#b = D\end{array}}{\widetilde{\mathcal{TE}} \vdash^{tdef} \underline{\texttt{struct S \{ } x : T_x \texttt{ \}}} : \bullet}$$

$$\text{[union]}\quad \frac{\begin{array}{l}\widetilde{\mathcal{TE}}(U) = \left\langle \begin{array}{c}\texttt{union U}\\ S\end{array}\right\rangle \\ \widetilde{\mathcal{TE}} \vdash^{type} T_x : \widetilde{\mathcal{TE}}(U)(x)\end{array}}{\widetilde{\mathcal{TE}} \vdash^{tdef} \texttt{union U \{ } x : T_x \texttt{ \}} : \bullet} \qquad \frac{\begin{array}{l}\widetilde{\mathcal{TE}}(U) = \left\langle \begin{array}{c}\texttt{union U}\\ D\end{array}\right\rangle \\ \widetilde{\mathcal{TE}} \vdash^{type} T_x : \widetilde{\mathcal{TE}}(U)(x), T_x\#b = D\end{array}}{\widetilde{\mathcal{TE}} \vdash^{tdef} \underline{\texttt{union U \{ } x : T_x \texttt{ \}}} : \bullet}$$

$$\text{[enum]}\quad \widetilde{\mathcal{TE}} \vdash^{tdef} \texttt{enum E \{ } x \texttt{ = } e \texttt{ \}} : \bullet$$

Figure 45: Binding time inference rules for type definitions

5.3.7 Well-annotated expressions

An annotated expression is said to be *well-annotated* if the binding time separation is consistent with the division of variables. This is captured by the relation

$\vdash^{exp}$: Expr $\times$ BType

defined in Figures 46 and 47.

Definition 5.8 *Suppose an annotated expression e and environments $\widetilde{\mathcal{E}}$ and $\widetilde{\mathcal{TE}}$ are given, such that $\widetilde{\mathcal{E}}$ is defined for all identifiers in e; it suits the underlying types, and it agree with $\widetilde{\mathcal{TE}}$. The expression e is* well-annotated *if there exists a type T s.t.*

$$\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e : T$$

where $\vdash^{exp}$ is defined in Figures 46 and 47. $\square$

The rules are justified as follows.

Constant and string-constants are static. A variable reference is never annotated dynamic, since all variables are bound to symbolic locations at specialization time, cf. Chapter 3.

The type of a struct member is given by the type environment. The struct indexing operator must be annotated dynamic if the struct cannot be split. The rules for pointer dereference and array indexing are similar in style. The type of value is determined by the value of the indirection[9] and, in the case of arrays, the index. If the index is dynamic, the indexing cannot be static.

Consider the rules for the address operator. If the subexpression is static, the result is a static pointer. Otherwise the application must be annotated dynamic, and the result is a dynamic pointer.[10]

Example 5.13 The rule for the address operator correctly captures applications such as '$\underline{\texttt{\&a[}e\texttt{]}}$' where e is a dynamic expression, but unfortunately also an expression such '$\underline{\texttt{\&a[2]}}$' where '`a`' is a array of dynamic values. By rewriting the latter expression into '`a + 2`' the problem can be circumvented. However, applications such as '`&x`' where '`x`' is dynamic inevitably become suspended. **End of Example**

[9]We assume that in an array expression $e_1[e_2]$, e_1 is of pointer type.

[10]See also Section 5.4.8 about function identifiers.

[const]	$\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \mathtt{c} : \left\langle \begin{smallmatrix} \tau_{\mathtt{b}} \\ S \end{smallmatrix} \right\rangle$	
[string]	$\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \mathtt{s} : \left\langle \begin{smallmatrix} * \\ S \end{smallmatrix} \right\rangle \left\langle \begin{smallmatrix} \mathtt{char} \\ S \end{smallmatrix} \right\rangle$	
[var]	$\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \mathtt{v} : \widetilde{\mathcal{E}}(\mathtt{v})$	
[struct]	$\dfrac{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : \left\langle \begin{smallmatrix} \mathtt{struct\ S} \\ S \end{smallmatrix} \right\rangle}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1.i : \widetilde{\mathcal{TE}}(S)(i)}$	$\dfrac{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : \left\langle \begin{smallmatrix} \mathtt{struct\ S} \\ D \end{smallmatrix} \right\rangle}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{e_1.i} : \widetilde{\mathcal{TE}}(S)(i)}$
[indr]	$\dfrac{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : \left\langle \begin{smallmatrix} * \\ S \end{smallmatrix} \right\rangle T_1}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} *e_1 : T_1}$	$\dfrac{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : \left\langle \begin{smallmatrix} * \\ D \end{smallmatrix} \right\rangle T_1}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{*e_1} : T_1}$
[array]	$\dfrac{\begin{array}{l}\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : \left\langle \begin{smallmatrix} * \\ S \end{smallmatrix} \right\rangle T_1 \\ \widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_2 : \left\langle \begin{smallmatrix} \tau_{\mathtt{b}} \\ S \end{smallmatrix} \right\rangle\end{array}}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1[e_2] : T_1}$	$\dfrac{\begin{array}{l}\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : \left\langle \begin{smallmatrix} * \\ D \end{smallmatrix} \right\rangle T_1 \\ \widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_2 : \left\langle \begin{smallmatrix} \tau_{\mathtt{b}} \\ B_2 \end{smallmatrix} \right\rangle\end{array}}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{e_1[e_2]} : T_1}$
[address]	$\dfrac{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1\#b = S}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \&e_1 : \left\langle \begin{smallmatrix} * \\ S \end{smallmatrix} \right\rangle T_1}$	$\dfrac{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1\#b = D}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{\&e_1} : \left\langle \begin{smallmatrix} * \\ D \end{smallmatrix} \right\rangle T_1}$
[unary]	$\dfrac{\begin{array}{l}\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1 \\ O(o) = (\left\langle \begin{smallmatrix} (T'_1) \\ \beta \end{smallmatrix} \right\rangle)T', T_1 \preceq \mathrm{stat}(T'_1)\end{array}}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} o\ e_1 : \mathrm{stat}(T')}$	$\dfrac{\begin{array}{l}\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1 \\ O(o) = \left\langle \begin{smallmatrix} (T'_1) \\ \beta \end{smallmatrix} \right\rangle T', T_1 \preceq \mathrm{dyn}(T'_1)\end{array}}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{o\ e_1} : \mathrm{dyn}(T')}$
[binary]	$\dfrac{\begin{array}{l}\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_i : T_i \\ O(o) = \left\langle \begin{smallmatrix} (T'_i) \\ \beta \end{smallmatrix} \right\rangle T', T_i \preceq \mathrm{stat}(T'_i)\end{array}}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1\ o\ e_2 : \mathrm{stat}(T')}$	$\dfrac{\begin{array}{l}\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_i : T_i \\ O(o) = \left\langle \begin{smallmatrix} (T'_i) \\ \beta \end{smallmatrix} \right\rangle T', T_i \preceq \mathrm{dyn}(T'_i)\end{array}}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{e_1\ o\ e_2} : \mathrm{dyn}(T')}$
[alloc]	$\dfrac{\widetilde{\mathcal{E}}(l) = \left\langle \begin{smallmatrix} * \\ S \end{smallmatrix} \right\rangle \left\langle \begin{smallmatrix} \mathtt{struct\ T} \\ S \end{smallmatrix} \right\rangle}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \mathtt{alloc}^l(T) : \left\langle \begin{smallmatrix} * \\ S \end{smallmatrix} \right\rangle \left\langle \begin{smallmatrix} \mathtt{struct\ T} \\ S \end{smallmatrix} \right\rangle}$	$\dfrac{\widetilde{\mathcal{E}}(l) = \left\langle \begin{smallmatrix} * \\ S \end{smallmatrix} \right\rangle \left\langle \begin{smallmatrix} \mathtt{struct\ T} \\ D \end{smallmatrix} \right\rangle}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{\mathtt{alloc}^l(T)} : \left\langle \begin{smallmatrix} * \\ D \end{smallmatrix} \right\rangle \left\langle \begin{smallmatrix} \mathtt{struct\ T} \\ D \end{smallmatrix} \right\rangle}$
[ecall]	$\dfrac{\begin{array}{l}\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} f : \left\langle \begin{smallmatrix} (T'_i) \\ B \end{smallmatrix} \right\rangle T_f \\ \widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_i : T_i, T_i\#b = S \\ f \text{ specified pure}\end{array}}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} f(e_1, \ldots, e_n) : \mathrm{stat}(T_f)}$	$\dfrac{\begin{array}{l}\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} f : \left\langle \begin{smallmatrix} (T'_i) \\ B \end{smallmatrix} \right\rangle T_f \\ \widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_i : T_i \\ T_i \preceq \mathrm{dyn}(T'_i)\end{array}}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{f(e_1, \ldots, e_n)} : \mathrm{dyn}(T_f)}$
[call]	$\dfrac{\begin{array}{l}\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_0 : \left\langle \begin{smallmatrix} (T'_i) \\ S \end{smallmatrix} \right\rangle T_0 \\ \widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_i : T_i \\ T_i \preceq T'_i, T \preceq T_0\end{array}}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_0(e_1, \ldots, e_n) : T}$	$\dfrac{\begin{array}{l}\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_0 : \left\langle \begin{smallmatrix} (T'_i) \\ D \end{smallmatrix} \right\rangle T'_0 \\ \widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_i : T_i \\ T_i \preceq T'_i, T'_0 \preceq T_0\end{array}}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{e_0(e_1, \ldots, e_n)} : T_0}$

Figure 46: Binding time inference rules for expressions (part 1)

The rules for unary and binary operator applications use the map O. If both arguments are static, a fresh instance of the operator's type is instantiated. Notice the usage of lift in the rule for dynamic applications to assure that possibly static values can be lifted. The rules for unary operator applications are analogous.

The result of an '`alloc`' call is given by the environment.[11]

The annotation of a call to an external function depends on whether the function is '`pure`'. In the affirmative case, a call where all parameters are static can be annotated static. Otherwise the call must be suspended.

[11]Consider the label of an alloc to be the name of a pointer variable.

[pre-inc]	$\dfrac{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1\#b = S}{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \texttt{++}e_1 : T_1}$	$\dfrac{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1\#b = D}{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{\texttt{++}e_1} : T_1}$
[post-inc]	$\dfrac{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1\#b = S}{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1\texttt{++} : T_1}$	$\dfrac{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1\#b = D}{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{e_1\texttt{++}} : T_1}$
[assign]	$\dfrac{\begin{array}{l}\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1\#b = S \\ \tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_2 : T_2, T_2\#b = S\end{array}}{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 \ aop \ e_2 : T_1}$	$\dfrac{\begin{array}{l}\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1\#b = D \\ \tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_2 : T_2, T_2 \preceq T_1\end{array}}{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{e_1 \ aop \ e_2} : T_1}$
[assign-se]	$\dfrac{\begin{array}{l}\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1\#b = S \\ \tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_2 : T_2, T_2\#b = ST_f\#b = S\end{array}}{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 \ aop \ e_2 : T_1}$	$\dfrac{\begin{array}{l}\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1\#b = D \\ \tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_2 : T_2, T_2 \preceq T_1\end{array}}{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{e_1 \ aop \ e_2} : T_1}$
[assign-dse]	$\dfrac{\begin{array}{l}\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1\#b = S \\ \tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_2 : T_2, T_2\#b = S \\ T_f\#b = S\end{array}}{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 \ aop \ e_2 : T_1}$	$\dfrac{\begin{array}{l}\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1\#b = D \\ \tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_2 : T_2, T_2 \preceq T_1\end{array}}{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{e_1 \ aop \ e_2} : T_1}$
[comma]	$\dfrac{\begin{array}{l}\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1\#b = S \\ \tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_2 : T_2, T_2\#b = S\end{array}}{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1, e_2 : T_2}$	$\dfrac{\begin{array}{l}\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1, T_1 \preceq \mathrm{dyn}(T_1) \\ \tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_2 : T_2, T_2 \preceq \mathrm{dyn}(T_2)\end{array}}{\tilde{\mathcal{E}}, TE \vdash^{exp} \underline{e_1, e_2} : \mathrm{dyn}(T_2)}$
[sizeof]		$\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{\texttt{sizeof}(T)} : \left\langle \begin{smallmatrix}\texttt{size\_t} \\ D\end{smallmatrix} \right\rangle$
[cast]	$\dfrac{\begin{array}{l}\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1 \\ \widetilde{\mathcal{TE}} \vdash^{type} T_e : T \\ \mathrm{Cast}(T_1, T), T1\#b = S\end{array}}{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} (T)e_1 : T}$	$\dfrac{\begin{array}{l}\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 : T_1 \\ \widetilde{\mathcal{TE}} \vdash^{type} T_e : T \\ \neg\mathrm{Cast}(T_1, T_e)\end{array}}{\tilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} \underline{(T)e_1} : T}$

Figure 47: Binding time inference rules for expressions (part 2)

The cases for calls to user-defined functions and indirect calls are treated as one for simplicity. The function designator must possess a function type, and the type of the actual must be lift-able to the formal argument types. In the case of a static application, all the parameters must be static.

The rules for pre- and post-increment expressions are straightforward. Consider the rules for assignments and recall that assignments to non-local objects and side-effects under conditional must be suspended. The pure syntactic criteria is conservative. In Section 5.4.8 we ameliorate the definition.

The first rules cover non-side-effecting assignments. The rules [assign-se] and [assign-dse] checks side-effecting assignments and conditional side-effecting assignments respectively.

We use T_f to denote the return type of the function containing the expression. Thus, if f contains the assignment, the type of f is $\left\langle \begin{smallmatrix}(T_i) \\ B\end{smallmatrix} \right\rangle T_f$.

The binding time of a comma expression depends on the binding time of both the expressions, and the implementation-defined '`sizeof`' special form must always be suspended. Finally, the rules for cast expressions are expressed via the predicate 'Cast' defined below. The type T_{e_1} is the type of the subexpression, and T_e is the new type.

Define 'Cast : Type × Type → Boolean' by

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathrm{Cast}(T_{from}, T_{to}) = \mathrm{case}(T_{from}, T_{to})} \\
\quad (\langle \tau_b' \rangle, \langle \tau_b'' \rangle) & = \mathrm{true} \\
\quad (\langle \langle * \rangle T', \langle * \rangle T'' \rangle) & = \mathrm{Cast}(T', T'') \\
\quad (\langle \langle \tau_b \rangle, \langle * \rangle T'' \rangle) & = \mathrm{false} \\
\quad (\langle \langle * \rangle T', \langle \tau_b \rangle \rangle) & = \mathrm{false}
\end{array}$$

compare with the analysis in Section 5.2.8.

Example 5.14 To allow partially-static operator applications, the following rule for binary plus in pointers and integers can be used.

$$\frac{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_i : T_i, T_i\#b = S}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{exp} e_1 \ +_{*int,int} \ e_2 : T_1}$$

where only the pointer specifier and the integer value is demanded static, not the indirection of the pointer. **End of Example**

5.3.8 Well-annotated statements

Well-annotatedness of a statement depends mainly on well-annotatedness of contained expressions. This is expressed by the means of the inference rules

$$\vdash^{stmt} : \mathrm{Stmt}$$

depicted in Figure 48.

Definition 5.9 *Let S be an annotated statement in a function f. Let $\widetilde{\mathcal{E}}$, $\widetilde{\mathcal{TE}}$ be a binding time environment and a type environment, such that $\widetilde{\mathcal{E}}$ is defined for all identifiers in S and $\widetilde{\mathcal{E}}$ agrees with $\widetilde{\mathcal{TE}}$. Then S is* well-annotated *if*

$$\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{stmt} S : \bullet$$

where $\vdash^{stmt}$ is defined in Figure 48. □

The rules are motivated as follows.

If a function contains a dynamic statement, the function must be residual, *i.e.* have a dynamic return type. This is reflected in the system by the side condition $T_f\#b = D$ in the dynamic rules.

An empty statement can always be annotated static, and the binding time of an expression statement depends on the expression. An `if` or `switch`[12] statement is dynamic if the test expression is dynamic. Furthermore, the sub-statements must be well-annotated.

Consider the rules for loops. The binding time of the loop construction depends on the test expression. The binding time of initializers does not necessarily influence the binding time of a loop, although it typically will be the case.

Finally, consider the rules for `return`. If the containing function is residual, the statement must be dynamic: a residual function must return a value at run-time — not at specialization time. In the case of base type values, the lift operator may be applied.

[12]For simplicity we assume 'break' and 'continue' are expressed via 'goto'.

$$\text{[empty]} \quad \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} ; : \bullet$$

$$\text{[expr]} \quad \frac{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e : T, T\#b = S}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} e : \bullet} \qquad \frac{\begin{array}{l}\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e : T, T\#b = D \\ T_f\#b = D\end{array}}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \underline{e} : \bullet}$$

$$\text{[if]} \quad \frac{\begin{array}{l}\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e : T, T\#b = S \\ \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_1 : \bullet \\ \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_2 : \bullet\end{array}}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \texttt{if (}e\texttt{) } S_1 \texttt{ else } S_2 : \bullet} \qquad \frac{\begin{array}{l}\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e : T, T\#b = D \\ \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_1 : \bullet \\ \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_2 : \bullet \\ T_f\#b = D\end{array}}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \underline{\texttt{if (}e\texttt{) } S_1 \texttt{ else } S_2} : \bullet}$$

$$\text{[switch]} \quad \frac{\begin{array}{l}\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e : T, T\#b = S \\ \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_1 : \bullet\end{array}}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \texttt{switch (}e\texttt{) } S_1 : \bullet} \qquad \frac{\begin{array}{l}\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e : T, T\#b = D \\ \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_1 : \bullet \\ T_f\#b = D\end{array}}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \underline{\texttt{switch (}e\texttt{) } S_1} : \bullet}$$

$$\text{[while]} \quad \frac{\begin{array}{l}\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e : T, T\#b = S \\ \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_1 : \bullet\end{array}}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \texttt{while (}e\texttt{) } S_1 : \bullet} \qquad \frac{\begin{array}{l}\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e : T, T\#b = D \\ \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_1 : \bullet \\ T_f\#b = D\end{array}}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \underline{\texttt{while (}e\texttt{) } S_1} : \bullet}$$

$$\text{[do]} \quad \frac{\begin{array}{l}\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e : T, T\#b = S \\ \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_1 : \bullet\end{array}}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \texttt{do } S_1 \texttt{ while (}e\texttt{)} : \bullet} \qquad \frac{\begin{array}{l}\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e : T, T\#b = D \\ \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_1 : \bullet \\ T_f\#b = D\end{array}}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \underline{\texttt{do } S_1 \texttt{ while (}e\texttt{)}} : \bullet}$$

$$\text{[for]} \quad \frac{\begin{array}{l}\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e_i : T_i, T_2\#b = S \\ \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_1 : \bullet \\ T_j\#b = D \Rightarrow T_f\#b = D, j = 1,3\end{array}}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \texttt{for (}e_1\texttt{;}e_2\texttt{;}e_3\texttt{) } S_1 : \bullet} \qquad \frac{\begin{array}{l}\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e_i : T_i, T_2\#b = D \\ \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_1 : \bullet \\ T_f\#b = D\end{array}}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \underline{\texttt{for (}e_1\texttt{;}e_2\texttt{;}e_3\texttt{) } S_1} : \bullet}$$

$$\text{[label]} \quad \frac{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_1 : \bullet}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} l\texttt{: } S_1 : \bullet}$$

$$\text{[goto]} \quad \widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \texttt{goto } m : \bullet$$

$$\text{[return]} \quad \frac{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e : T, T\#b = S, T_f\#b = S}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \texttt{return } e : \bullet} \qquad \frac{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{exp} e : T, T_f\#b = D, T \preceq T_f}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \underline{\texttt{return } e} : \bullet}$$

$$\text{[block]} \quad \frac{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} S_i : \bullet}{\widetilde{\mathcal{E}},\widetilde{\mathcal{TE}} \vdash^{stmt} \{S_i\} : \bullet}$$

Figure 48: Binding-time inference rules for statements

5.3.9 Well-annotated functions

Recall that we solely consider functions that are annotated for specialization and (possibly) shared. Well-annotatedness of a function depends on the consistency between the parameters and local variables, and the statements. This is expressed in the rules

$\vdash^{fun}$: Fun

defined in Figure 49.

$$\text{[share]}\quad \frac{\begin{array}{l}\widetilde{\mathcal{E}}(f) = \left\langle {}^{(BT_i)}_{\;S} \right\rangle BT_f, d_i \equiv x_i : T_i \\ \widetilde{\mathcal{TE}} \vdash^{decl} x_j : T_j \;:\; BT_j, d_j \equiv x_j : T_j \\ \widetilde{\mathcal{E}}[x_i \mapsto BT_i, x_j \mapsto BT_j], \widetilde{\mathcal{TE}} \vdash^{stmt} S_k : \bullet \\ BT_i\#b = S, \text{statdyn}(BT_i) \\ BT_f\#b = S, \text{statdyn}(BT_f)\end{array}}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{fun} \langle T_f, d_i, d_j, S_k\rangle : \bullet} \qquad \frac{\begin{array}{l}\widetilde{\mathcal{E}}(f) = \left\langle {}^{(BT_i)}_{\;B} \right\rangle BT_f, d_i \equiv x_i : T_i \\ \widetilde{\mathcal{TE}} \vdash^{decl} x_j : T_j \;:\; BT_j, d_j \equiv x_j : T_j \\ \widetilde{\mathcal{E}}[x_i \mapsto BT_i, x_j \mapsto BT_j], \widetilde{\mathcal{TE}} \vdash^{stmt} S_k : \bullet \\ \text{statdyn}(BT_i) BT_f\#b = D, \text{statdyn}(BT_f)\end{array}}{\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{fun} \underline{\langle T_f, d_i, d_j, S_k\rangle} : \bullet}$$

Figure 49: Binding-time inference rules for functions

Definition 5.10 *Let f be a function in a program p. Let $\widetilde{\mathcal{E}}$ be a binding type environment defined on all global identifiers in p, and $\widetilde{\mathcal{TE}}$ a type environment for p with which $\widetilde{\mathcal{E}}$ agrees.*

The function f is well-annotated *if*

$$\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{fun} f : \bullet$$

where $\vdash^{fun}$ is defined in Figure 49. □

The rules can be explained as follows. The type of the function is given by the bt-type environment which includes the type of the parameters. The bt-type of local variables is determined by the means of the inference rules for definitions. In the environment extended with bt-types for parameters and locals, the consistency of the annotation of statements is checked.

5.3.10 Well-annotated C programs

The well-annotatedness of a C program depends on type definitions, global declarations, and function definitions. The type definitions define a type environment which the bt-type environment for variables must agree with. The functions must be well-annotated in the context of the type and binding time environment.

Definition 5.11 *Let $p = \langle \mathcal{T}, \mathcal{D}, \mathcal{F}\rangle$ be an annotated program. It is* well-annotated *provided*

1. *$\widetilde{\mathcal{TE}}$ is a type environment s.t. $\widetilde{\mathcal{TE}} \vdash^{tdef} t : \bullet$, $t \in \mathcal{T}$;*
2. *$\widetilde{\mathcal{E}}$ is a binding time environment defined for all identifiers in p, s.t. $\widetilde{\mathcal{TE}} \vdash^{decl} d : \widetilde{\mathcal{E}}(x)$, where $d \equiv x : T_x \in \mathcal{D}$, and*
3. *$\widetilde{\mathcal{E}}, \widetilde{\mathcal{TE}} \vdash^{fun} f : \bullet$, $f \in \mathcal{F}$,*

cf. Definitions 5.7, 5.6 and 5.10. □

Theorem 5.1 *A* well-annotated *program fulfills the congruence principle and the requirements for the generating extension transformation.*

Proof By the justification given for the inference systems. □

5.4 Binding time inference

The previous section specified conditions for a program to be well-annotated. The problem of binding-time analysis is the opposite: given an initial division of the input parameters, to find a well-annotated version of the program. Naturally, by making all constructs dynamic, an well-annotated version can easily be found,[13] but we desire "most-static" annotation.

The analysis we present is constraint based and works as follows. First a constraint system is *collected* by a single traversal of the program's syntax tree. The constraints capture the dependencies between an expression and its subexpressions. Next, the system is *normalized* by a set of rewrite rules. We show that the rules are normalizing, and hence that constraint systems have a normal form. Finally, a *solution* is found. It turns out that solving a normalized constraint system is almost trivial.

Given a solution, it is easy to derive a well-annotated program. We show that the annotation is *minimal* in the sense that all other well-annotations will make more constructs dynamic.

5.4.1 Constraints and constraint systems

We give a constraint-based formulation of the inference systems presented in the previous section. A *constraint system* is a multi-set of formal constraints of the form

$T_1 = T_2$	Equal constraint
$T_1 \preceq T_2$	Lift constraint
$B_1 \rhd B_2$	Dependency constraint

where T_i range over bt-types and B_i range over binding times. In addition, the types T_1 and T_2 in a constraint $T_1 \preceq T_2$ must agree on the underlying static program types.[14] For instance, a constraint $\left\langle {\texttt{int} \atop S} \right\rangle \preceq \left\langle {* \atop S} \right\rangle \left\langle {\texttt{double} \atop D} \right\rangle$ illegal. We use $\mathcal{C}$ to range over constraint systems.

Define $\prec^* \in \text{BType} \times \text{BType}$, $\rhd^* \in \text{BTime} \times \text{BTime}$:

$\left\langle {\tau_b \atop S} \right\rangle$	$\prec^*$	$\left\langle {\tau_b \atop D} \right\rangle$	Lift base
D	$\rhd^*$	D	Dynamic dependency
S	$\rhd^*$	B	No dependency

where τ_b range over base type specifiers, and B range over binding times. Define $T_1 \preceq^* T_2 \overset{def}{\Longleftrightarrow} T_1 \preceq T_2$ or $T_1 = T_2$.

Intuitively, the lift constraint corresponds to the lift $<$ relation used in the previous chapter; the dependency constraint captures "if B_1 is dynamic then B_2 must also be dynamic".

Definition 5.12 *Let $\mathcal{C}$ be a constraint system. A* solution *to $\mathcal{C}$ is a substitution S :* BTVar $\to$ BTime *such that for all constraint $c \in \mathcal{C}$:*

[13]Obviously, constants should be kept static.

[14]This requirement will be fulfilled by the construction of constraint systems.

$$\begin{array}{ll} ST_1 = ST_2 & \textit{if } c \equiv T_1 = T_2 \\ ST_1 \preceq^* ST_2 & \textit{if } c \equiv T_1 \preceq T_2 \\ SB_1 \rhd^* SB_2 & \textit{if } c \equiv B_1 \rhd B_2 \end{array}$$

where application of substitution is denoted by juxtaposition, and S is the identify on all binding time variable not appearing in $\mathcal{C}$. The set of solutions is denoted by $\mathrm{Sol}(\mathcal{C})$. $\Box$

Let binding times be ordered by $S \sqsubset D$ and extend point-wise to solutions. Obviously, if a constraint system has a solution it has a *minimal solution* that maps more binding time variables to S than all other. The constraint systems of interest all have at least one solution, and thus a minimal solution.

Example 5.15 The system

$$\{\left\langle {\texttt{int} \atop B_1} \right\rangle \preceq \left\langle {\texttt{int} \atop B_2} \right\rangle, \left\langle {* \atop B_3} \right\rangle \left\langle {\texttt{int} \atop B_4} \right\rangle \preceq \left\langle {* \atop B_5} \right\rangle \left\langle {\texttt{int} \atop B_6} \right\rangle, D \rhd B_3\}$$

has the solution $S = [B_1, B_2 \mapsto S, B_3, B_4, B_5, B_6 \mapsto D]$, and it is minimal. The system $\{\left\langle {\texttt{int} \atop B_1} \right\rangle \preceq \left\langle {\texttt{int} \atop S} \right\rangle, D \rhd B_1\}$ has no solution. **End of Example**

5.4.2 Binding time attributes and annotations

Attribute all program types with fresh binding time variables, *e.g.* if '`p`' has type $\langle * \rangle \langle \texttt{int} \rangle$, its attributed type is $\left\langle {* \atop \beta_1} \right\rangle \left\langle {\texttt{int} \atop \beta_2} \right\rangle$. In practice, this step is be done during parsing. The aim of the analysis is to instantiate the binding time variables consistently. An instantiation of the variables is given by a substitution $S : \mathrm{BTVar} \to \{S, D\}$.

For a substitution S, let '$\mathrm{Ann}_S(p)$' be an *annotation function* that underlines program constructs in accordance with the instantiation. For example, if e is an array index '$e_1[e_2]$' and e_1 has the instantiated type $\left\langle {* \atop D} \right\rangle \left\langle {\texttt{int} \atop D} \right\rangle$, the index shall be underlined. We omit formal specification. The key point is to observe that the inference rules stating well-annotatedness are deterministic on the bt-types. Overload Ann to declarations, expressions, statements and functions.

The previous section employed a binding time environment $\widetilde{\mathcal{E}}$ for mapping an identifier to its bt-type. However, when we assume that all variable definitions (and declarations) are assigned unique bt-types, the mapping $\widetilde{\mathcal{E}}(x) = T_x$ is implicit in the program. In the following we write T_x for the bt-type of a definition (of) x.[15] If f is a function identifier, we write $\left\langle {(T_i) \atop B} \right\rangle T_f$ for its type; notice that T_f denotes the type of the returned value.

Example 5.16 A (partially) attributed program ('`strindex`'; Kernighan and Ritchie page 69) is shown in Figure 50. All expressions and variable definitions are annotated with their type. The bt-type $\left\langle {* \atop \beta_{si}} \right\rangle \left\langle {\texttt{char} \atop \beta{*}si} \right\rangle$ equals $T_{strindex}$. **End of Example**

Similarly, the type environment $\widetilde{\mathcal{TE}}$ is superfluous; for a struct S we write T_S for the bt-type of S ($= \widetilde{\mathcal{TE}}(S) \downarrow 1$), and T_{S_x} for the bt-type of field x ($= \widetilde{\mathcal{TE}}(S)(x)$).

Given an attributed definition $x : T_x$ we say that the environment $\widetilde{\mathcal{E}}$ agrees with if $\widetilde{\mathcal{E}}(x) = T_x$. Similarly for type environments. Application of a substitution to an environment is denoted by juxtaposition.

[15]Assume for ease of presentation unique names.

$$
\begin{array}{l}
\textit{/* strindex: return first position of p in s */}\\
\texttt{char *:}\left\langle\begin{smallmatrix}*\\ \beta\_si\end{smallmatrix}\right\rangle\left\langle\begin{smallmatrix}\texttt{char}\\ \beta\_{*}si\end{smallmatrix}\right\rangle\texttt{strindex(char *p:}\left\langle\begin{smallmatrix}*\\ \beta\_p\end{smallmatrix}\right\rangle\left\langle\begin{smallmatrix}\texttt{char}\\ \beta\_{*}p\end{smallmatrix}\right\rangle\texttt{, char *s:}\left\langle\begin{smallmatrix}*\\ \beta\_s\end{smallmatrix}\right\rangle\left\langle\begin{smallmatrix}\texttt{char}\\ \beta\_{*}s\end{smallmatrix}\right\rangle\texttt{)}\\
\texttt{\{}\\
\quad\texttt{int k:}\left\langle\begin{smallmatrix}\texttt{int}\\ \beta\_k\end{smallmatrix}\right\rangle\texttt{;}\\
\quad\texttt{for (; * != '\textbackslash 0'; s++) \{}\\
\qquad\texttt{for (k = 0; p[k] != '\textbackslash 0' \&\& p[k] == s[k]; k++);}\\
\qquad\texttt{if (p:}\left\langle\begin{smallmatrix}\texttt{char}\\ \beta\_1\end{smallmatrix}\right\rangle\left\langle\begin{smallmatrix}\texttt{char}\\ \beta\_2\end{smallmatrix}\right\rangle\texttt{[k:}\left\langle\begin{smallmatrix}\texttt{int}\\ \beta\_3\end{smallmatrix}\right\rangle\texttt{]:}\left\langle\begin{smallmatrix}\texttt{char}\\ \beta\_4\end{smallmatrix}\right\rangle\texttt{ == '\textbackslash 0':}\left\langle\begin{smallmatrix}\texttt{char}\\ \beta\_5\end{smallmatrix}\right\rangle\texttt{) return s:}\left\langle\begin{smallmatrix}*\\ \beta\_6\end{smallmatrix}\right\rangle\left\langle\begin{smallmatrix}\texttt{char}\\ \beta\_7\end{smallmatrix}\right\rangle\texttt{;}\\
\quad\texttt{\}}\\
\quad\texttt{return NULL:}\left\langle\begin{smallmatrix}*\\ \beta\_8\end{smallmatrix}\right\rangle\left\langle\begin{smallmatrix}\texttt{char}\\ \beta\_9\end{smallmatrix}\right\rangle\texttt{;}\\
\texttt{\}}
\end{array}
$$

Figure 50: Binding-time type annotated program

$$
\begin{array}{lll}
\text{[decl]} & \dfrac{\vdash^{ctype} T_1 : \bullet}{\vdash^{cdecl} \texttt{extern}\ x : T : \bullet} & \{D\ \triangleright T\#b\}\\
\text{[pure]} & \dfrac{\vdash^{ctype} T_1 : \bullet}{\vdash^{cdecl} \texttt{pure extern}\ x : T_1 : \bullet} & \{\}\\
\text{[def]} & \dfrac{\vdash^{ctype} T_1 : \bullet}{\vdash^{cdecl} x : T_1\ \ : \bullet} & \{\}\\
\text{[res]} & \dfrac{\vdash^{ctype} T_1 : \bullet}{\vdash^{cdecl} \texttt{residual}\ x\ \ : T : \bullet} & \{D\ \triangleright T\#b\}\\
\text{[base]} & \vdash^{ctype} \left\langle\begin{smallmatrix}\tau_{\texttt{b}}\\ \beta\end{smallmatrix}\right\rangle : \bullet & \{\}\\
\text{[struct]} & \widetilde{\mathcal{TE}} \vdash^{ctype} \left\langle\begin{smallmatrix}\texttt{struct S}\\ \beta\end{smallmatrix}\right\rangle : \bullet & \{\left\langle\begin{smallmatrix}\texttt{struct S}\\ \beta\end{smallmatrix}\right\rangle = T_S\}\\
\text{[ptr]} & \dfrac{\vdash^{ctype} T_1 : \bullet}{\vdash^{ctype} \left\langle\begin{smallmatrix}*\\ \beta\end{smallmatrix}\right\rangle T_1 : \bullet} & \{\beta\ \triangleright T_1\#b\}\\
\text{[array]} & \dfrac{\vdash^{ctype} T_1 : \bullet}{\vdash^{ctype} \left\langle\begin{smallmatrix}\texttt{[n]}\\ \beta\end{smallmatrix}\right\rangle T_1 : \bullet} & \{\beta\ \triangleright T_1\#b\}\\
\text{[fun]} & \dfrac{\vdash^{cdel} x_i : T_i : \bullet, d_i \equiv x : T_i \quad \vdash^{ctype} T : \bullet}{\vdash^{ctype} \left\langle\begin{smallmatrix}(d_i)\\ \beta\end{smallmatrix}\right\rangle T : \bullet} & \{\beta\ \triangleright T\#b, T_i\#b\ \triangleright \beta\} \cup \bigcup_i \mathrm{statdyn}(T_i)
\end{array}
$$

Figure 51: Constraint-based binding-time inference for declarations

5.4.3 Capturing binding times by constraints

This section gives a constraint-based characterization of well-annotated programs. The reformulation of the inference systems of the previous section is almost straightforward. For each expression, constraints connecting the bt-types of the expression and its subexpressions are generated, and furthermore, constraints capturing global dependencies, *e.g.* that a `return` statement in a dynamic function must be dynamic, are added. We prove below that a solution to a constraint system corresponds to a well-annotated program.

Declarations and type definitions

The set of constraints generated for a declaration is defined inductively in Figure 51.

Observe that the rules are deterministic and no value, besides constraints, is "returned". Thus, the constraints can be generated by a traversal over the syntax tree where constraints are accumulated in a global data structure.

The function 'statdyn', used by the rule for function types, is defined as follows.

For a bt-type $\left\langle {\tau_1 \atop \beta_1} \right\rangle \ldots \left\langle {\tau_n \atop \beta_n} \right\rangle$, it returns the set of constraints $\bigcup_i \{\beta_{i+1} \rhd \beta_i\}$. If a type specifier $\left\langle {\texttt{struct S} \atop \beta} \right\rangle$, occurs in T, statdyn(T) adds the dependencies $\{T_{S_x}\#b \rhd T_S\#b\}$ for all members x of S.

Intuitively, given a type T, if the constraints statdyn(T) are added to the constraint set, a solution must either map all binding times in T to dynamic or to static. In the case of struct types, the dependencies assure that if one member is dynamic, the struct becomes dynamic (forcing all members to be dynamic).

Example 5.17 Assume the struct definition '`struct S { int x, int y }` s'. We have

$$\text{statdyn}(T_s) = \{T_{S_x}\#b \rhd T_S\#b, T_{S_y}\#b \rhd T_S\}$$

Assume that y is dynamic, that is $T_{S_y}\#b = D$. Then $T_S\#b$ must be dynamic, and also T_{S_x} (the latter follows from the constraints generated for type definitions, see below). This is used for preventing partially static parameters. **End of Example**

Lemma 5.3 *Let $d \in$ Decl in a program p. The constraint system for d as defined by Figure 51 has a* minimal solution S_0. *Let $\widetilde{\mathcal{TE}}$ be an environment that agrees with p. Then*

$$S_0(\widetilde{\mathcal{TE}}) \vdash^{decl} Ann_{S_0}(d) : S_0(T_d)$$

where T_d is the bt-type of d.

The lemma says that d is well-annotated under the solution to the constraint system.

Proof To see that a solution exists, notice that the substitution that maps all variables to D is a solution.

Suppose that $d \equiv x : T$. Consider first the type T. Proof by induction on the number of type specifiers. In the base case, T is either a base type specifiers or a struct type specifier. The well-annotatedness follows from the (unspecified) definition of Ann. In the inductive case, (ptr, array, fun), the cases for [ptr] and [array] are obvious. Notice the dependency constraints guarantee that the bt-types are well-formed. Consider the rule for function types. The second dependency constraint makes sure that if one parameter is dynamic, a solution must map the binding time variable β to dynamic, as required.

Well-annotatedness of d follows easily. □

Suppose now that T is a type definition. The constraint-based formulation of binding time inference is stated in Figure 52.

Lemma 5.4 *Let $T \in$ TDef in a program p. The constraint system for T as defined by Figure 52 has a* minimal solution S_0. *Let $\widetilde{\mathcal{TE}}$ be an environment that agrees with p. Then*

$$S_0(\widetilde{\mathcal{TE}}) \vdash^{tdef} Ann_{S_0}(T) : \bullet$$

Proof By inspection of the rules. □

$$
\begin{array}{lll}
[\text{struct}] & \dfrac{\vdash^{ctype} T_x : \bullet}{\vdash^{ctdef} \texttt{struct S \{ } x : T_x \texttt{ \} } : \bullet} & \{T_S\#b \rhd T_x\#b\} \\
[\text{union}] & \dfrac{\vdash^{ctype} T_x : \bullet}{\vdash^{ctdef} \texttt{union U \{ } x : T_x \texttt{ \} } : \bullet} & \{T_U\#b \rhd T_x\#b\} \\
[\text{enum}] & \vdash^{ctdef} \texttt{enum E \{ } x = e \texttt{ \} } : \bullet & \{\}
\end{array}
$$

Figure 52: Constraints generated for type definition

$$
\begin{array}{lll}
[\text{const}] & \vdash^{cexp} c : \left\langle \genfrac{}{}{0pt}{}{\tau_{\mathtt{b}}}{\beta} \right\rangle & \{\left\langle \genfrac{}{}{0pt}{}{\tau_{\mathtt{b}}}{\beta} \right\rangle = \left\langle \genfrac{}{}{0pt}{}{\tau_{\mathtt{b}}}{S} \right\rangle\} \\
[\text{string}] & \vdash^{cexp} s : \left\langle \genfrac{}{}{0pt}{}{*}{\beta} \right\rangle \left\langle \genfrac{}{}{0pt}{}{\texttt{char}}{\beta_1} \right\rangle & \{\left\langle \genfrac{}{}{0pt}{}{*}{\beta} \right\rangle \left\langle \genfrac{}{}{0pt}{}{\texttt{char}}{\beta_1} \right\rangle = \left\langle \genfrac{}{}{0pt}{}{*}{\beta} \right\rangle \left\langle \genfrac{}{}{0pt}{}{\texttt{char}}{S} \right\rangle\} \\
[\text{var}] & \vdash^{cexp} v : T & \{T = T_v\} \\
[\text{struct}] & \dfrac{\vdash^{cexp} e_1 : \left\langle \genfrac{}{}{0pt}{}{\texttt{struct S}}{\beta_1} \right\rangle}{\vdash^{cexp} e_1.i : T} & \{\left\langle \genfrac{}{}{0pt}{}{\texttt{struct S}}{\beta_1} \right\rangle = T_S, T = T_{S_i}\} \\
[\text{indr}] & \dfrac{\vdash^{cexp} e_1 : \left\langle \genfrac{}{}{0pt}{}{*}{\beta_1} \right\rangle T_1}{\vdash^{cexp} *e_1 : T} & \{T = T_1\} \\
[\text{array}] & \dfrac{\vdash^{cexp} e_1 : \left\langle \genfrac{}{}{0pt}{}{*}{\beta_1} \right\rangle T_1 \quad \vdash^{cexp} e_2 : \left\langle \genfrac{}{}{0pt}{}{\tau_{\mathtt{b}}}{\beta_2} \right\rangle}{\vdash^{cexp} e_1[e_2] : T} & \{T = T_1, \beta_2 \rhd \beta_1\} \\
[\text{address}] & \dfrac{\vdash^{cexp} e_1 : T_1}{\vdash^{cexp} \&e_1 : \left\langle \genfrac{}{}{0pt}{}{*}{\beta} \right\rangle T} & \{T = T_1, T_1\#b \rhd \beta\} \\
[\text{unary}] & \dfrac{\vdash^{cexp} e_1 : T_1, T_1' = \overline{T_1} \quad \vdash^{ctype} T : \bullet}{\vdash^{cexp} o\ e_1 : T} & \{T_1 \preceq T_1', T = T', T_1\#b \rhd T\#b, T\#b \rhd T_1'\#b\} \\
[\text{binary}] & \dfrac{\vdash^{cexp} e_i : T_i, \overline{T_i'} = T_i \quad \vdash^{ctype} T : \bullet}{\vdash^{cexp} e_1\ o\ e_2 : T} & \{T_i \preceq T_i', T_i\#b \rhd T\#b, T\#b \rhd T_i'\#b\} \\
[\text{alloc}] & \vdash^{cexp} \texttt{alloc}^l(S) : T & \{T = T_l\} \\
[\text{ecall}] & \dfrac{\vdash^{cexp} e_i : T_i, T_i' = \overline{T_i}}{\vdash^{cexp} f(e_1, \ldots, e_n) : T} & \{T_i \preceq T_i', T\#b \rhd T_i'\#b, T_i\#b \rhd T\#b\} \\
[\text{call}] & \dfrac{\vdash^{cexp} e_i : T_i \quad \vdash^{cexp} e_0 : \left\langle \genfrac{}{}{0pt}{}{(i : T_i')}{\beta} \right\rangle T'}{\vdash^{cexp} e_0(e_1, \ldots, e_n) : T} & \{T_i \preceq T_i', T' \preceq T\}
\end{array}
$$

Figure 53: Constraint-based binding-time inference for expressions (part 1)

Expressions

The inference system for expressions is given in the Figure 53 and 54. Recall that T_v denotes the bt-type assigned to v.

Observe that the value-flow analysis described in Chapter 2 renders unification of the fields in a struct assignment '`s = t`' superfluous. The types of '`s`' and '`t`' are equal. In practice this is a major gain since unification of struct types is expensive.

The rules for unary and binary operator applications, and extern function application relates the type T of an argument with an equivalent type $\overline{T}$ (denoting a "fresh" instance of T). This implements lifting of arguments. Notice that it is the over-lined type that is suspended if the application is dynamic.

The rule for casts uses a function Cast to generate constraints, defined as follows.

$$
\begin{array}{lll}
\text{[pre-inc]} & \dfrac{\vdash^{cexp} e_1 : T_1}{\vdash^{cexp} \texttt{++e}_1 : T} & \{T = T_1\} \\[2ex]
\text{[post-inc]} & \dfrac{\vdash^{cexp} e_1 : T_1}{\vdash^{cexp} e_1\texttt{++} : T} & \{T = T_1\} \\[2ex]
\text{[assign]} & \dfrac{\vdash^{cexp} e_1 : T_1 \quad \vdash^{cexp} e_2 : T_2}{\vdash^{cexp} e_1 \; aop \; e_2 : T} & \{T_2 \preceq T_1, T = T_1\} \\[2ex]
\text{[assign-se]} & \dfrac{\vdash^{cexp} e_1 : T_1 \quad \vdash^{cexp} e_2 : T_2}{\vdash^{cexp} e_1 \; aop^{\dagger} \; e_2 : T} & \{T_2 \preceq T_1, T = T_1, T_f\#b \rhd T_1\#b\} \\[2ex]
\text{[assign-cse]} & \dfrac{\vdash^{cexp} e_1 : T_1 \quad \vdash^{cexp} e_2 : T_2}{\vdash^{cexp} e_1 \; aop \; ^{\ddagger} \; e_2 : T} & \{T_2 \preceq T_1, T = T_1, T_f\#b \rhd T_1\#b\} \\[2ex]
\text{[comma]} & \dfrac{\vdash^{cexp} e_1 : T_1 \quad \vdash^{cexp} e_2 : T_2}{\vdash^{cexp} e_1, e_2 : T} & \{T_2 \preceq T, T_1\#b \rhd T_2\#b\} \\[2ex]
\text{[sizeof]} & \vdash^{cexp} \texttt{sizeof}(T) : T & \{T = \left\langle \substack{\texttt{size\_t} \\ D} \right\rangle\} \\[2ex]
\text{[cast]} & \dfrac{\vdash^{cexp} e_1 : T_1}{\vdash^{cexp} (\texttt{T})e_1 : T} & \mathrm{Cast}(T_1, T)
\end{array}
$$

Figure 54: Constraint-based binding-time inference for expressions (part 2)

$$
\begin{array}{ll}
\mathrm{Cast}(T_{from}, T_{to}) = \mathrm{case}(T_{from}, T_{to}) & \\
\quad (\left\langle \substack{\tau_b' \\ \beta_1} \right\rangle, \left\langle \substack{\tau_b'' \\ \beta_2} \right\rangle) & = \{\beta_1 \rhd \beta_2\} \\
\quad (\left\langle \substack{* \\ \beta_1} \right\rangle T', \left\langle \substack{* \\ \beta_2} \right\rangle T'') & = \{\beta_1 \rhd \beta_2\} \cup \mathrm{Cast}(T', T'') \\
\quad (\left\langle \substack{\tau_b \\ \beta_1} \right\rangle, \left\langle \substack{* \\ \beta_2} \right\rangle T'') & = \{D \rhd \beta_2\} \\
\quad (\left\langle \substack{* \\ \beta_1} \right\rangle T', \left\langle \substack{\tau_b \\ \beta_2} \right\rangle) & = \{D \rhd \beta_2\}
\end{array}
$$

Notice the similarity with the definition in Section 5.3.

Lemma 5.5 *Let $e \in \mathrm{Expr}$ in a program p. The constraint system for e as defined by Figures 53 and 54 has a* minimal solution *S_0. Let $\widetilde{\mathcal{E}}$, $\widetilde{\mathcal{TE}}$ be environments that agree with p. Then*

$$S_0(\widetilde{\mathcal{E}}), S_0(\widetilde{\mathcal{TE}}) \vdash^{exp} Ann_{S_0}(e) : S_0(T_e)$$

where T_e is the bt-type of e.

Proof The constraint system clearly has a solution. Proof by structural induction.

The base cases [const], [string] and [var] follow immediately.

The inductive cases are easy to check. For example, case [array]: Suppose that the solution maps β_1 to S. Due to the constraint $\beta_2 \rhd \beta_1$, β_2 is must be S. By the induction hypothesis, e_1 and e_2 are well-annotated, and the expression is a well-annotated static array expression. Now suppose that the solution maps β_1 to D. Underlining the array index makes the expression well-annotated. □

[empty]	$\vdash^{cstmt} ; : \bullet$	$\{\}$
[exp]	$\dfrac{\vdash^{cexp} e : T}{\vdash^{cstmt} e : \bullet}$	$\{T\#b \rhd T_f\#b\}$
[if]	$\dfrac{\vdash^{cexp} e : T \quad \vdash^{cstmt} S_1 : \bullet \quad \vdash^{cstmt} S_2 : \bullet}{\vdash^{cstmt} \texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2 : \bullet}$	$\{T\#b \rhd T_f\#b\}$
[switch]	$\dfrac{\vdash^{cexp} e : T \quad \vdash^{cstmt} S_1 : \bullet}{\vdash^{cstmt} \texttt{switch (}e\texttt{)}\ S_1 : \bullet}$	$\{T\#b \rhd T_f\#b\}$
[case]	$\dfrac{\vdash^{cstmt} S_1 : \bullet}{\vdash^{cstmt} \texttt{case}\ e\texttt{:}\ S_1 : \bullet}$	$\{\}$
[default]	$\dfrac{\vdash^{cstmt} S_1 : \bullet}{\vdash^{cstmt} \texttt{default:}\ S_1 : \bullet}$	$\{\}$
[while]	$\dfrac{\vdash^{cexp} e : T \quad \vdash^{cstmt} S_1 : \bullet}{\vdash^{cstmt} \texttt{while (}e\texttt{)}\ S_1 : \bullet}$	$\{T\#b \rhd T_f\#b\}$
[do]	$\dfrac{\vdash^{cexp} e : T \quad \vdash^{cstmt} S_1 : \bullet}{\vdash^{cstmt} \texttt{do}\ S_1\ \texttt{while (}e\texttt{)} : \bullet}$	$\{T\#b \rhd T_f\#b\}$
[for]	$\dfrac{\vdash^{cexp} e_i : T_i \quad \vdash^{cstmt} S_1 : \bullet}{\vdash^{cstmt} \texttt{for (}e_1\texttt{;}e_2\texttt{;}e_3\texttt{)}\ S_1 : \bullet}$	$\{T_i\#b \rhd T_f\#b\}$
[label]	$\dfrac{\vdash^{cstmt} S_1 : \bullet}{\vdash^{cstmt} l : S_1 : \bullet}$	$\{\}$
[goto]	$\vdash^{cstmt} \texttt{goto}\ m : \bullet$	$\{\}$
[return]	$\dfrac{\vdash^{cexp} e : T}{\vdash^{cstmt} \texttt{return}\ d}$	$\{T \preceq T_f\}$
[block]	$\dfrac{\vdash^{cstmt} S_i : \bullet}{\vdash^{cstmt} \{S_i\} : \bullet}$	$\{\}$

Figure 55: Constraint-based inference rules for statements

Statements

Let s be a statement in a function f. Recall that T_f denotes the return type of f. The inference rules and constraints for statements are displayed in Figure 55.

Lemma 5.6 *Let $s \in \mathrm{Stmt}$ in a function f in a program p. The constraint system corresponding to s, as defined by Figure 55, has a* minimal solution S_0. *Let $\widetilde{\mathcal{E}}$ and $\widetilde{\mathcal{TE}}$ be environments that agree with p. Then*

$$S_0(\widetilde{\mathcal{E}}), S_0(\widetilde{\mathcal{TE}}) \vdash^{stmt} Ann_{S_0}(S) : \bullet$$

Proof It it easy to see that the constraint system has a solution. The proof of well-annotatedness is established by structural induction on statements.

The only interesting case is [return]. Suppose that the solution maps $T_f\#b$ to S. Due to the constraint $T \preceq T_f$, the binding of the return value must be static. On the other hand, if the function is dynamic by the solution, the constraint assures that the value is lift-able, as required by the well-annotatedness rules. $\Box$

$$[\text{share}]\ \frac{\vdash^{cdecl} d_j : \bullet \vdash^{cstmt} S_k : \bullet}{\vdash^{cfun} \langle T_f, d_i, d_j, S_k\rangle : \bullet} \quad \text{statdyn}(d_i)$$

Figure 56: Constraint-based binding time inference for functions

Functions

Figure 56 defines the constraint generated for functions.

Lemma 5.7 *Let $f \in$ Fun in a program p. The constraint system for f as defined in Figure 56 has a* minimal solution S_0. *Let $\widetilde{\mathcal{E}}$ and $\widetilde{\mathcal{TE}}$ be environments that agree with p. Then*

$$S_0(\widetilde{\mathcal{E}}), S_0(\widetilde{\mathcal{TE}}) \vdash^{fun} Ann_{S_0}(f) : \bullet$$

Proof By inspection of the rule and Lemmas 5.3 and 5.6. □

Program

The constraint definition for a program is defined by the following definition.

Definition 5.13 *Let $p \equiv \langle \mathcal{T}, \mathcal{D}, \mathcal{F}\rangle$ be a program. Let $\widetilde{\mathcal{TE}}$ be a type environment that agree with $\mathcal{T}$. Define:*

1. *$\mathcal{C}_t$ to be the constraint system generated due to $\vdash^{ctdef} t : \bullet$ for all $t \in \mathcal{T}$.*
2. *$\mathcal{C}_d$ to be the constraint system generated due to $\vdash^{cdecl} d : T$ for all $d \in \mathcal{D}$.*
3. *$\mathcal{C}_f$ to be the constraint system generated due to $\vdash^{cfun} f : \bullet$ for all $f \in \mathcal{F}$.*
4. *$\mathcal{C}_0 = \{D \rhd T_{x_d}\}$, where x_d are the dynamic variables in the initial division.*

The constraint system for p is then

$$\mathcal{C}_{pgm}(p) = \mathcal{C}_t \cup \mathcal{C}_d \cup \mathcal{C}_f \cup \mathcal{C}_0$$

□

Theorem 5.2 *Let p be a program. The constraint system $\mathcal{C}_{pgm}(p)$ has a minimal solution S_0. Then $Ann_{S_0}(p)$ is a* well-annotated *program.*

Proof Follows from Lemmas 5.4, 5.3 and 5.7. □

We state without proof that a minimal solution gives a "best" annotation, *i.e.* when an initial division is fixed, it is not possible to construct "more" static well-annotation.

5.4.4 Normal form

This section presents a set of solution preserving rewrite rules that simplify the structure of constraint systems. This allows a solution to be found easily

Let $\mathcal{C}$ be a constraint system. By $\mathcal{C} \Rightarrow^{S} \mathcal{C}'$ we denote the application of a rewrite rule that yields system $\mathcal{C}'$ under substitution S. The notation $\mathcal{C} \Rightarrow *^{S}\mathcal{C}'$ denotes exhaustive application of rewrite rules until the system stabilizes. We justify below that definition is meaningful, *i.e.* that a system eventually will converge.

A rewrite rule is *solution preserving* if $\mathcal{C} \Rightarrow^{S'} \mathcal{C}'$ and $S \in \mathrm{Sol}(\mathcal{C}) \Leftrightarrow S \circ S' \in \mathrm{Sol}(\mathcal{C}')$. That is, a solution to the transformed system $\mathcal{C}'$ composed with the substitution S' is solution to the original system. Suppose that a constraint system is rewritten exhaustively $\mathcal{C} \Rightarrow *^{S'}\mathcal{C}'$, and $\mathcal{S}_0$ is a minimal solution to $\mathcal{C}'$. We desire $S_0 \circ S'$ to be a minimal solution to $\mathcal{C}$.

Figure 57 defines a set of rewrite rules for binding time constraints, where T range over bt-types and B range over binding times. The function unify : BType $\times$ BType $\rightarrow$ BTVar $\rightarrow$ BTime denotes the must general unifier over binding time types (actually, binding times). Notice that Unify never can fail (in this case), and that no rule discards a variables (*e.g.* due to a rule $\mathcal{C} \cup \{S \;\triangleright \beta\} \Rightarrow \mathcal{C}$).

Lemma 5.8 *The rewrite rules displayed in Figure 57 are solution preserving.*

Proof By case analysis.
Case 1.a: Follows from the definition of Unify.
Case 2.d: Follows from the definition of $\preceq$.
Case 2.g: Left to right: Suppose S_0 is a solution to $\left\langle {}^{\tau_{\mathsf{b}}}_{\beta} \right\rangle \preceq \left\langle {}^{\tau_{\mathsf{b}}}_{\beta'} \right\rangle$. If $S_0(\beta) = D$, then $S_0(\beta') = D$, by definition of $\preceq$. By definition, S_0 is also a solution to $\beta \triangleright \beta'$. Suppose $S_0(\beta) = S$. Then $S_0(\beta')$ either S or D, but then S_0 is also a solution to the right hand side. Right to left: Similar.
Cast 3.c: A solution to the left hand side maps β to S or D, and solves the constraint on the right hand side. □

Lemma 5.9 *The rewrite rules in Figure 57 are normalizing.*

Proof All rules but 1.g, 1.h, 1.i, 2.f, 2.g, 2.h, 2.i, 2.j, 2.k, 3.c, 3.f and 3.g remove a constraint. The constraints introduced by rule 1.g are never rewritten. The constraint introduced by rules 2.h, 2.i, and 2.j are removed by rules 1. The constraint added by rule 2.f is not subject for other rules. The constraints introduced by rules 3.c, 3.f and 3.g are either left in the system or removed by rules 2.a and 2.b. The constraint introduced by rule 2.g is either removed (directly or indirectly) or left in the system. Consider rules 1.h, 1.i and 2.k. The number of times these rules can be applied is limited by the size of bt-types. Notice that a constraint introduced by a rule cannot be subject for the same rule again. □

The lemma proves that exhaustive application of the rewrite rules in Figure 57 is well-defined. Let $\mathcal{C}$ be a constraint system, and suppose $\mathcal{C} \Rightarrow *^S \mathcal{C}'$. The system $\mathcal{C}'$ is in *normal form*; a normal form is not unique. The theorem below characterizes normal form constraint systems.

Theorem 5.3 *Let $\mathcal{C}$ be a constraint system.*

1. *The system $\mathcal{C}$ has a normal form, and it can be found by exhaustive application $\mathcal{C} \Rightarrow *^{S'} \mathcal{C}'$ of the rewrite rules in Figure 57.*
2. *A normal form constraint system consists of constraints of the form:*

$$\left\langle {\tau_{\mathsf{b}} \atop S} \right\rangle \preceq \left\langle {\tau_{\mathsf{b}} \atop \beta} \right\rangle, \beta \rhd \beta'$$

   and no other constraints.

3. *If S_0' is a minimal solution to $\mathcal{C}'$, then $S = S_0' \circ S'$ is a minimal solution to $\mathcal{C}$.*

Proof **Case 1**: Follows from Lemma 5.9.
Case 2: By inspection of the rules in Figure 57.
Case 3: Suppose that S_0' is a minimal solution to $\mathcal{C}'$. Observe that for a solution S to $\mathcal{C}$, if $S'(\beta) = D$ then $S(\beta) = D$, since otherwise a constraint would not be satisfied. This implies that $S_0' \circ S'$ is a minimal solution to $\mathcal{C}$. □

The theorem states that a minimal solution to a constraint system can be found as follows: first normalize the constraint system, and then find a minimal solution to the normalized constraint system. The composition of the substitutions is a minimal solution.

5.4.5 Solving constraints

Finding a minimal solution to a normal form constraint system is notably simple: solve all lift constraints by equality, and map remaining free variables to S.

Lemma 5.10 *Let $\mathcal{C}'$ be a normal form constraint system. The substitution $S = [\beta \mapsto S]$ for all $\beta \in \mathit{FreeVar}(\mathcal{C}')$ is a minimal solution to $\mathcal{C}'$.*

Proof To see that S is a solution, suppose the opposite. Then there exists a constraint that is not satisfied by S. Due to the characterization of normal form, this constraint must be of the form $\left\langle {\tau \atop S} \right\rangle \preceq \left\langle {\tau \atop \beta} \right\rangle$ or $\beta \rhd \beta'$. However, these are solved when all variables are mapped to S.

Clearly, S is the minimal solution. □

Given this, the following theorem states a constructive procedure for binding-time analysis.

Equal			
1.a	$\mathcal{C} \cup \{\left\langle {\tau \atop S} \right\rangle = \left\langle {\tau \atop S} \right\rangle\}$	$\Rightarrow \mathcal{C}$	
1.b	$\mathcal{C} \cup \{\left\langle {\tau \atop D} \right\rangle = \left\langle {\tau \atop D} \right\rangle\}$	$\Rightarrow \mathcal{C}$	
1.c	$\mathcal{C} \cup \{\left\langle {\tau \atop \beta} \right\rangle = \left\langle {\tau \atop S} \right\rangle\}$	$\Rightarrow S\mathcal{C}$	$S = [\beta \mapsto S]$
1.d	$\mathcal{C} \cup \{\left\langle {\tau \atop \beta} \right\rangle = \left\langle {\tau \atop D} \right\rangle\}$	$\Rightarrow S\mathcal{C}$	$S = [\beta \mapsto D]$
1.e	$\mathcal{C} \cup \{\left\langle {\tau \atop S} \right\rangle = \left\langle {\tau \atop \beta} \right\rangle\}$	$\Rightarrow S\mathcal{C}$	$S = [\beta \mapsto S]$
1.f	$\mathcal{C} \cup \{\left\langle {\tau \atop D} \right\rangle = \left\langle {\tau \atop \beta} \right\rangle\}$	$\Rightarrow S\mathcal{C}$	$S = [\beta \mapsto D]$
1.g	$\mathcal{C} \cup \{\left\langle {\tau \atop \beta} \right\rangle = \left\langle {\tau \atop \beta'} \right\rangle\}$	$\Rightarrow S\mathcal{C} \cup \{\left\langle {\mathtt{int} \atop S} \right\rangle \preceq \left\langle {\mathtt{int} \atop \beta} \right\rangle\}$	$S = [\beta' \mapsto \beta]$
1.h	$\mathcal{C} \cup \{\left\langle {\tau \atop B} \right\rangle BT = \left\langle {\tau \atop B'} \right\rangle BT'\}$	$\Rightarrow \mathcal{C} \cup \{\left\langle {\tau \atop B} \right\rangle = \left\langle {\tau \atop B'} \right\rangle, BT = BT'\}$	
1.i	$\mathcal{C} \cup \{\left\langle {(BT_i) \atop B} \right\rangle BT = \left\langle {(BT'_i) \atop B'} \right\rangle\}$	$\Rightarrow \mathcal{C} \cup \{BT_i = BT'_i, \left\langle {\mathtt{int} \atop B} \right\rangle = \left\langle {\mathtt{int} \atop B'} \right\rangle, BT = BT'\}$	
Lift			
2.a	$\mathcal{C} \cup \{\left\langle {{}^{\tau}\mathtt{b} \atop S} \right\rangle \preceq \left\langle {{}^{\tau}\mathtt{b} \atop S} \right\rangle\}$	$\Rightarrow \mathcal{C}$	
2.b	$\mathcal{C} \cup \{\left\langle {{}^{\tau}\mathtt{b} \atop S} \right\rangle \preceq \left\langle {{}^{\tau}\mathtt{b} \atop D} \right\rangle\}$	$\Rightarrow \mathcal{C}$	
2.c	$\mathcal{C} \cup \{\left\langle {{}^{\tau}\mathtt{b} \atop D} \right\rangle \preceq \left\langle {{}^{\tau}\mathtt{b} \atop D} \right\rangle\}$	$\Rightarrow \mathcal{C}$	
2.d	$\mathcal{C} \cup \{\left\langle {{}^{\tau}\mathtt{b} \atop D} \right\rangle \preceq \left\langle {{}^{\tau}\mathtt{b} \atop \beta} \right\rangle\}$	$\Rightarrow S\mathcal{C}$	$S = [\beta \mapsto D]$
2.e	$\mathcal{C} \cup \{\left\langle {{}^{\tau}\mathtt{b} \atop \beta} \right\rangle \preceq \left\langle {{}^{\tau}\mathtt{b} \atop S} \right\rangle\}$	$\Rightarrow S\mathcal{C}$	$S = [\beta \mapsto S]$
2.f	$\mathcal{C} \cup \{\left\langle {{}^{\tau}\mathtt{b} \atop \beta} \right\rangle \preceq \left\langle {{}^{\tau}\mathtt{b} \atop D} \right\rangle\}$	$\Rightarrow \mathcal{C} \cup \{\left\langle {{}^{\tau}\mathtt{b} \atop S} \right\rangle \preceq \left\langle {{}^{\tau}\mathtt{b} \atop \beta} \right\rangle\}$	
2.g	$\mathcal{C} \cup \{\left\langle {{}^{\tau}\mathtt{b} \atop \beta} \right\rangle \preceq \left\langle {{}^{\tau}\mathtt{b} \atop \beta'} \right\rangle\}$	$\Rightarrow \mathcal{C} \cup \{\beta \rhd \beta'\}$	
2.h	$\mathcal{C} \cup \{\left\langle {{}^{\tau}\mathtt{s} \atop B} \right\rangle \preceq \left\langle {{}^{\tau}\mathtt{s} \atop B'} \right\rangle\}$	$\Rightarrow \mathcal{C} \cup \{\left\langle {{}^{\tau}\mathtt{s} \atop B} \right\rangle = \left\langle {{}^{\tau}\mathtt{s} \atop B'} \right\rangle\}$	
2.i	$\mathcal{C} \cup \{\left\langle {* \atop B} \right\rangle BT \preceq \left\langle {* \atop B'} \right\rangle BT'\}$	$\Rightarrow \mathcal{C} \cup \{\left\langle {* \atop B} \right\rangle BT = \left\langle {* \atop B'} \right\rangle BT'\}$	
2.j	$\mathcal{C} \cup \{\left\langle {[\mathtt{n}] \atop B} \right\rangle BT \preceq \left\langle {[\mathtt{n}] \atop B'} \right\rangle BT'\}$	$\Rightarrow \mathcal{C} \cup \{\left\langle {[\mathtt{n}] \atop B} \right\rangle BT = \left\langle {[\mathtt{n}] \atop B'} \right\rangle BT'\}$	
2.k	$\mathcal{C} \cup \{\left\langle {(BT_i) \atop B} \right\rangle BT \preceq \left\langle {(BT'_i) \atop B'} \right\rangle BT'\}$	$\Rightarrow \mathcal{C} \cup \{BT'_i = BT_i, BT = BT', B \rhd B'\}$	
Dependency			
3.a	$\mathcal{C} \cup \{S \rhd S\}$	$\Rightarrow \mathcal{C}$	
3.b	$\mathcal{C} \cup \{S \rhd D\}$	$\Rightarrow \mathcal{C}$	
3.c	$\mathcal{C} \cup \{S \rhd \beta\}$	$\Rightarrow \mathcal{C} \cup \{\left\langle {\mathtt{int} \atop S} \right\rangle \preceq \left\langle {\mathtt{int} \atop \beta} \right\rangle\}$	
3.d	$\mathcal{C} \cup \{D \rhd D\}$	$\Rightarrow \mathcal{C}$	
3.e	$\mathcal{C} \cup \{D \rhd \beta'\}$	$\Rightarrow S\mathcal{C}$	$S = [\beta' \mapsto D]$
3.f	$\mathcal{C} \cup \{\beta \rhd S\}$	$\Rightarrow \mathcal{C} \cup \{\left\langle {\mathtt{int} \atop S} \right\rangle \preceq \left\langle {\mathtt{int} \atop \beta} \right\rangle\}$	
3.g	$\mathcal{C} \cup \{\beta \rhd D\}$	$\Rightarrow \mathcal{C} \cup \{\left\langle {\mathtt{int} \atop S} \right\rangle \preceq \left\langle {\mathtt{int} \atop \beta} \right\rangle\}$	

Figure 57: Normalizing rewrite rules

Theorem 5.4 *Let $\mathcal{C}$ be a constraint system. The minimal solution S_0 to $\mathcal{C}$ is given by $S_0 = S_0' \circ S'$, where $\mathcal{C}'$ is a normal form of $\mathcal{C}$: $\mathcal{C} \Rightarrow *^{S'}\mathcal{C}'$, and S_0' maps all free variables in $\mathcal{C}'$ to S.*

Proof The substitution S_0' is a minimal solution to $\mathcal{C}'$ according to Lemma 5.10. Due to Theorem 5.3, item 3, S is a minimal solution to $\mathcal{C}$. □

5.4.6 Doing binding-time analysis

The steps in the binding-time analysis can be recapitulated as follows. Let p be a program.

1. Construct the constraint system $\mathcal{C}_{pgm}$ as defined by Definition 5.13.
2. Normalize the constraint system to obtain a normal form $\mathcal{C}'$ by exhaustive application of the rewrite rules in Figure 57, under substitution S'.
3. Let S_0' be the substitution that maps all variables in $\mathcal{C}'$ to S.

Then $S = S_0' \circ S'$ is a minimal solution. Apply the annotation function $\mathrm{Ann}_S(p)$ to get a well-annotated program.

Step 1 can be done during a single traversal of the program's syntax tree. By interpreting un-instantiate binding time variables as S, step 3 can be side-stepped. Thus, to get an efficient binding-time analysis, construction of an efficient normalization algorithm remain.

5.4.7 From division to well-annotated program

The analysis developed here assigns to each expression its bt-type. This implies that for instance the generating extension transformation (Chapter 3) can transform an expression looking solely at the expression's type, since the well-annotatedness of an expression is determined solely by the binding times of its subexpressions. This is, however, in general excessively much information. At the price of more computation during the transformation, a *division* is sufficient.

A *division* is a map from identifiers to bt-types.[16] By propagating information from subexpressions to expressions, the binding time of all constructs can be found.

This implies that once a bijection between identifiers and their associated bt-types has been established, the constraint set can be kept and solved completely separately from the program. The solution assigns binding times to identifiers, from which the binding times of all expressions can be determined. This fact is exploited in Chapter 7 that considers separate binding-time analysis.

[16]We assume included in the set of identifiers the labels of 'alloc()' calls.

5.4.8 Extensions

Recall from Chapter 3 that conditional side-effects under dynamic control must be annotated dynamic. The rule in Figure 54 suspends all conditional side-effects (which is correct but conservative). If the test is static, there is no need to suspend the side-effect (in a non-sharable function). This can be incorporated by adding a dependency constraint from the test to the assignment: $\{B_{test} \rhd B_{assign}\}$.

The rule for unions does not implement the sharing of initial members of struct members of a union. To correct this, constraints unifying the binding times of the relevant members must be added, such that all initial members possess the same binding time. We will not dwell with a formal definition.

Recall from Chapter 3 that unfold-able functions may be assigned more static binding times than sharable functions. For example, partially-static parameters of pointer type can be allowed, since propagation of non-local variables is not risked. The extension is almost trivial, but tedious to describe.[17]

Finally, consider the rule for the address operator in Figure 46. To implement specialization to function pointers, it must be changed slightly. Recall that an application '`&f`', where '`f`' is a function designator, should be classified static. This case can easily be detected by the means of the static types.

5.5 Efficient constraint normalization algorithm

This section presents an efficient algorithm for constraint normalization, the core part of the binding-time analysis. The algorithm is based on a similar algorithm originally by Henglein [Henglein 1991], but is simpler due to the factorization into a value-flow analysis (for structs) and the binding-time analysis, and the exploitation of static types.

5.5.1 Representation

The normalization rules in Figure 57 rewrite several constraints into "trivial" lift constraints of the form: $\left\langle \begin{smallmatrix}\mathtt{int}\\ S\end{smallmatrix} \right\rangle \preceq \left\langle \begin{smallmatrix}\mathtt{int}\\ \beta\end{smallmatrix} \right\rangle$. The objective being that otherwise a variable could be discarded from the constraint system, and hence not included in the domain of a solution. This is solely a technical problem; in practice bt-types are attached to expressions and do not "disappear". Since, by Theorem 5.4, un-instantiated binding time variables are to going to be mapped to S after the normalization anyway, it is safe to skip trivially satisfied constraints.

To every binding time variable β we assign a list of dependent binding times. If β is a binding time variable, β_{dep} is the list of binding time variables β' such that $\beta \rhd \beta'$.

For unification of binding times, we employ a variant of union/find [Tarjan 1983]. In the algorithms only union on binding time terms are performed, although we for notational simplicity assume that '`find()`' also works on bt-types (returning the ECR of the binding time of the first type specifier).

[17]We have not implemented this in *C-Mix*.

```
/* Efficient constraint normalization algorithm */
for (c in clist)
   switch (c) {
      case BT1 = BT2: /* Equality constraint */
         union_type(BT1,BT2);
      case BT1 <= BT2: /* Lift constraint */
         switch (find(BT1), find(BT2)) {
            case (<base,S>,<base,S>): case (<base,D>,<base,D>):
            case (<base,S>,<base,b>): case (<base,b>,<base,D>): /* L1 */
               break;
            case (<base,S>,<base,D>): break;                    /* L2 */
            case (<base,b>,<base,S>): union(b,S); break;        /* L3 */
            case (<base,D>,<base,b>): union(b,D); break;        /* L4 */
            case (<base,b1>,<base,b2>):                         /* L5 */
               b2.dep = add_dep(b1,b2.dep); break;
            case (<struct,B1>,<struct,B2>):                     /* L6 */
               union(B1,B2); break;
            default: /* ptr, array and fun type: unify */
               union_type(BT1,BT2); break;
         }
      case B1 |> B2: /* Dependency constraint */
         switch (find(B1), find(B2)) {
            case (S,S): case (S,D): case (D,D):                 /* D1 */
            case (S,b): case (b,S): case (b,D): break;
            case (D,b): union(b,D); break;                      /* D2 */
            case (b1,b2): b2.dep = add_dep(b1,b2.dep); break;   /* D3 */
         }
   }
/* equal: union BT1 and BT2 */
void union_type(BType BT1, BType BT2)
{
   for ( (bt1,bt2) in (BT1,BT2) )
      switch (find(bt1),find(bt2)) {
         case (<(BT_i'),B1>,<(BT_i''),B2>):
            for ( (BT1,BT2) in (BT_i',BT_i'') )
               union_type(BT1,BT2);
            union(find(B1),find(B2));
            break;
         default:
            union(find(bt1), find(bt2)); break;
       }
 }
```

Figure 58: Efficient constraint normalization algorithm

```
/* union: unify simple (ECR) terms b1 and b2 */
void union(BTime b1, BTime b2)
{
   switch (b1,b2) {
      case (S,S): case (D,D): break;
      case (b,S): case (S,b): b = link(S); break;
      case (b,D): case (D,b):
         dep = b.dep; b = link(D);
         for (b' in dep) union(find(b'),D);
         break;
      case (b1,b2):
        b1.dep = add_dep(b1.dep,b2.dep);
        b2 = link(b1);
        break;
   }
}
```

Figure 59: Union algorithm adopted to normalization algorithm (without rank)

5.5.2 Normalization algorithm

The normalization algorithm is depicted in Figure 58. The input is a list '`clist`' of constraints. The algorithm side-effects the type representation such that all binding time variables in a minimal solution that would map to dynamic are instantiated to D.

In the case of an equality constraint, the type specifiers are union-ed. Notice that no "type-error" can occur: the underlying static program types match.

The trivial lift constraints are skipped. In the case $\left\langle \genfrac{}{}{0pt}{}{\tau_b}{\beta} \right\rangle \preceq \left\langle \genfrac{}{}{0pt}{}{\tau_b}{S} \right\rangle$ and $\left\langle \genfrac{}{}{0pt}{}{\tau_b}{D} \right\rangle \preceq \left\langle \genfrac{}{}{0pt}{}{\tau_b}{\beta} \right\rangle$ where β is forced to be either S or D, respectively, the binding time variables are union-ed. In the case of a constraint $\left\langle \genfrac{}{}{0pt}{}{\tau_b}{\beta_1} \right\rangle \preceq \left\langle \genfrac{}{}{0pt}{}{\tau_b}{\beta_2} \right\rangle$, the dependency from β_1 to β_2 is recorded in the dependency list of β_1. If the type is not a base type, the binding times of the two binding time types shall be equal, which is accomplished via the case for equality constraint.

The dependency list of a binding time variable is checked by the union function before the variable is made dynamic, as shown in Figure 59. For simplicity we ignore the maintenance of ranks.

5.5.3 Complexity

Consider the algorithm in Figure 58. Clearly, the algorithm possesses a run-time that is linear in the number of constraints. Observe that no constraints are added to the constraint list during the processing. The '`union_type()`' does not, however, take constant time. In the case of function types, the arguments must be processed.

The amortized run-time of the analysis is almost-linear in the size of the constraint system, which is linear in the size of the program. To this, the complexity of the value-flow analysis (for structs) must be added.

In practice, the number of constraints generated for every node is close to 1 on the average, see the benchmarks provided in Section 5.7. Notice, however, that the implementation is optimized beyond the present description.

5.5.4 Correctness

The correctness of the algorithm amounts to showing that it implements the rewrite rules in Figure 57. We provide an informal argument.

The rules for equality constraints are captured by the '`union_type()`' function. We omit a proof of its correctness.

Consider the rules for lift. Rules 2.d and 2.e are covered by L3 and L4, respectively. Rule 2.g is implemented by L5, and rule 2.b by L2. Rules 2.a, 2.c and 2.f are captured by L1 (trivial constraints). Case L6 corresponds to rule 2.h. The rules 2.i, 2.j and 2.k are implemented by the default case.

Finally, the rules for dependency constraints. Rules 3.a, 3.b, 3.c, 3.d, 3,f and 3.g are trivial and are implemented by case D1. Rule 3.e is implemented by D2, and case D3 converts the constraint $\beta \rhd \beta'$ into the internal representation (corresponding to the rewrite rules that leave a constraints in the system).

Since the '`union()`' algorithm makes dynamic all dependent variables when a variable becomes dynamic, the normalization algorithm correspond to exhaustive application of the rewrite rules.

5.5.5 Further improvements

Even though the binding-time analysis is efficient there is room for improvements. The normalization algorithm is linear in the number of constraints. The easiest way to lower the run-time of the binding-time analysis is to reduce the number of constraints!

Consider the constraints generated for binary operator applications. Lift constraints are added to capture that an operand possibly may be lifted. However, in the case of values of struct or pointer types, no lifting can take place. Thus, equality constraints can be generated instead.

The key point is that by inspection of the static program types, several lift constraints can be replaced by equality constraints, which are faster to process.

Next, the analysis' storage usage is linear in the number of constraints. It can be reduced by *pre-normalization* during the generation. For example, all equality constraints can be processed immediately. This reduces the number of constraints to about the half. Practical experiments show that this is a substantial improvement, even though it does not improve the overall algorithm's complexity.

5.6 Polyvariant binding-time analysis

The analysis developed so far is *monovariant* on function arguments. A parameter is assigned one binding time only, approximating all calls in the program to the function. A *polyvariant* binding-time analysis is context-sensitive; different calls to a function are not (always) collapsed. In this section we outline a polyvariant analysis based on the same principles as employed in Chapter 4.

5.6.1 Polyvariant constraint-based analysis

We describe a polyvariant analysis based on program's static-call graph. Recall that the static-call graph approximates context-sensitive invocations of functions. A function called in n different contexts is said to have n variants. The static-call graph maps a call and a variant to a variant of the called function, see Chapter 2.

To each type specifier appearing in a function with n variants we assign a vector of $n+1$ binding time variables. The variables 1,...,n describe the binding times of the variants, and variable 0 is a summary (corresponding to monovariant analysis). The summary is also used for indirect calls.[18]

Example 5.18 Consider the program in Example 5.1. The initial binding time assignment to '`pow`' is

$$\left\langle \genfrac{}{}{0pt}{}{\left(\left\langle \genfrac{}{}{0pt}{}{\texttt{int}}{\langle\beta^0_{\texttt{n}},\beta^1_{\texttt{n}},\beta^2_{\texttt{n}}\rangle} \right\rangle, \left\langle \genfrac{}{}{0pt}{}{\texttt{int}}{\langle\beta^0_{\texttt{x}},\beta^1_{\texttt{x}},\beta^2_{\texttt{x}}\rangle} \right\rangle\right)}{\langle\beta_0,\beta_1,\beta_2\rangle} \right\rangle \left\langle \genfrac{}{}{0pt}{}{\texttt{int}}{\left\langle\beta^0_{pow},\beta^1_{pow},\beta^2_{pow}\right\rangle} \right\rangle$$

where the length of the vectors are 3 since '`pow`' appears in two contexts. **End of Example**

The constraint generation proceeds as in the intra-procedural case except for calls and `return` statements.[19]

Consider a call $g^l(e_1,\ldots,e_n)$ in a function with n variants. Suppose $\mathcal{SCG}(l,i) = \langle g, k^i\rangle$. The constraints generated are

$$\bigcup_{i=1,\ldots,n} \{T^i_j \preceq T^{k^i}_{g_j}, T^{k^i}_g \preceq T^i\}$$

where $T^{k^i}_{g_j}$ denotes the bt-type of the j'th parameter in the i'th variant, and $T^{k^i}_g$ the return type.

Example 5.19 For the '`pow()`' function we have: $T^1_n = \left\langle \genfrac{}{}{0pt}{}{\texttt{int}}{\beta^1_n} \right\rangle$. **End of Example**

For `return` statements, the following constraints are generated:

$$\bigcup_{i=1,\ldots,n} \{T^i \preceq T^{k^i}_g\}$$

relating the binding time of the i'th variant with the function's bt-type.

Finally, constraints

$$\bigcup_{i=1,\ldots,n} \{T^i_t\# \rhd T^0_t\#\}$$

for all type specifiers t are added. These constraints causes variant 0 to be a summary variant.

It is straightforward to extend the normalization algorithm in Figure 58 to inter-procedural analysis. The vectors of binding time variables are processed component-wise.

[18]Recall that the static-call graph does not approximate indirect calls.

[19]The constraint generating for assignments must also be changed to accommodate inter-procedural side-effects under dynamic control.

Example 5.20 The result of inter-procedural analysis of the program in Example 5.1 is as follows.

$$\left\langle \begin{array}{c} (\left\langle \begin{array}{c}\texttt{int}\\ \langle D,S,D\rangle\end{array}\right\rangle, \left\langle \begin{array}{c}\texttt{int}\\ \langle D,D,S\rangle\end{array}\right\rangle) \\ \langle D,S,D\rangle \end{array} \right\rangle \left\langle \begin{array}{c} \texttt{int} \\ \langle D,D,D\rangle \end{array} \right\rangle$$

where the result value is dynamic in both cases. **End of Example**

We refer to Chapter 4.6 for a detailed description of the technique.

5.6.2 Polyvariance and generation extensions

The polyvariant binding time information can be exploited by the generating extension transformation developed in Chapter 3 as follows.

For every function, a generating function is generated for each variant and the 0 variant. If one or more contexts have the same binding times signature, they can be collapsed. Copying of functions is also known as *procedure cloning* [Cooper *et al.* 1993]. At the time of writing we have not implemented function cloning on the basis of binding times into the *C-Mix* system.

5.7 Examples

We have implemented the binding-time analysis in the *C-Mix* system. The analysis is similar to the one describe in this chapter, but optimized. Foremost, constraints are pre-normalized during generation. For example, most equality constraints are unified immediately.

We have timed the analysis on some test programs. The experiments were carried out on a Sun SparcStation II with 64 Mbytes of memory. The results are shown in the table below. See Chapter 9 for a description of the test programs.

Program	Lines	Constraints	Normalization	Analysis
Gnu strstr	64	148	≈ 0.0 sec	0.03 sec
Ludcmp	67	749	0.02 sec	0.04 sec
Ray tracer	1020	8,241	0.4 sec	0.7 sec
ERSEM	≈ 5000	112,182	5.5 sec	8.7 sec

As can be seen the analysis is *very* fast. Notably is the seemingly non-linear relationship between the number of constraints for the Ray tracer and the ERSEM modeling system. The reason is that ERSEM contains many array indexing operators giving rise to lift constraints, whereas more constraints can be pre-normalized in the case of the ray tracer.

5.8 Related work

Binding-time analysis was originally introduced into partial evaluation as a means for obtaining efficient self-application of specializers. The use of off-line approximation of binding times as opposed to online determination includes several other advantages, however. It yields faster specializers, enables better control over the desired degree of specialization, and can provide useful feedback about prospective speedups, binding-time improvements and, more broadly, the binding time separation in a program. Furthermore, it can guide program transformations such as the generation extension transformation.

5.8.1 BTA by abstract interpretation

The binding-time analysis in the first Mix was based on abstract interpretation over the domain $S \sqsubset D$ [Jones *et al.* 1989]. It coarsely classified data structures as either completely static or completely dynamic, invoking the need for manual binding time improvements. By the means of a closure analysis, Bondorf extended the principles to higher-order Scheme [Bondorf 1990].

To render manual binding time improvement superfluous, Mogensen developed a binding-time analysis for partially static data structures [Mogensen 1989]. The analysis describes the binding time of data structures by the means of a tree grammar. Launchbury has developed a projection-based analysis that in a natural way captures partially static data structures [Launchbury 1990].

All the mentioned analyses are monovariant and for applicative languages.

Ruggieri *et al.* have developed a lifetime analysis for heap-allocated objects, to replace dynamically allocated objects by variables [Ruggieri and Murtagh 1988]. The analysis classify objects as compile time or run time, and is thus similar to binding-time analysis. It is based on classical data-flow analysis methods.

5.8.2 BTA by type inference and constraint-solving

The concept of two-level languages was invented by the Nielsons who also developed a binding-time analysis for a variant of the lambda calculus [Nielson and Nielson 1988]. The analysis was partly based on abstract interpretation and partly on type inference via Algorithm W. Gomard designed a binding-time analysis for an untyped lambda calculus using a backtracking version of Algorithm W [Gomard 1990]. The analysis is conjectured to run in cubic time.

Henglein reformulated the problem and gave a constraint-based characterization of binding time inference [Henglein 1991]. Further, he developed an efficient constraint normalization algorithm running in almost-linear time.

In our Master's Thesis, we outlined a constraint-based binding-time analysis for a subset of C [Andersen 1991], which later was considerably simplified and implemented [Andersen 1993a]. This chapter provides a new formulation exploiting the static types, and adds polyvariance.

Bondorf and Jørgensen have re-implemented the analyses in the Similix Scheme partial evaluator to constraint-based analyses [Bondorf and Jørgensen 1993], and Birkedal and Welinder have developed a binding-time analysis for the Core part of Standard ML [Birkedal and Welinder 1993].

Heintze develops the framework of set-based analysis in his thesis [Heintze 1992]. Binding-time analysis can be seen as a instance of general set-based analysis. It is suggested that polyvariant analysis can be obtained by copying of functions' constraint sets.

5.8.3 Polyvariant BTA

Even though polyvariant binding-time analysis is widely acknowledged as a substantial improvement of a partial evaluator's strength only little work has been done (successfully).

Rytz and Gengler have extended the (original) binding-time analysis in Similix to a polyvariant version by iteration of the original analysis [Rytz and Gengler 1992]. Expressions that may get assigned two (incomparable) binding times are duplicated, and the analysis started from scratch. Naturally, this is very expensive in terms of run-time.

Consel has constructed a polyvariant analysis for the Schism partial evaluation, that treats a higher-order subset of Scheme [Consel 1993a]. The analysis is based on abstract interpretation and uses a novel concept of filters to control the degree of polyvariance.

A different approach has been taken by Consel and Jouvelot by combining type and effect inference to obtain a polyvariant binding-time analysis [Consel and Jouvelot 1993]. Currently, the analysis can only handle non-recursive programs, and no efficient algorithm has been developed.

Henglein and Mossin have developed a polyvariant analysis based on polymorphic type inference [Henglein and Mossin 1994]. The idea is to parameterize types over binding times. For example, a lambda expression $\lambda x : \tau.e$ may be assigned the "type scheme" $\Lambda\beta.\tau_1 \rightarrow^S \tau_2$, where β denotes the binding time of x (and appears in the type τ_2), and the S on the function arrow symbol denotes 'static closure'.

5.9 Further work

This section list a number of topics for further study.

5.9.1 Constraint solving, tracing and error messages

We have seen that a clean separation of constraint *generation* and *solving* is advantageous: the program needs only be traversed once and, as shown in Chapter 7, this supports separate analysis. The separation renders useful feedback from the constraint-solver hard, however. The problem is that once the constraint system has been extracted, the connection to the program is lost. In the case of binding-time analysis, the solver will never fail, as *e.g.* a constraint-based type checker might do, but *traces* of value flow would be useful. Obvious questions are "what forced this variable to be dynamic", and "why do '`x`' and '`y`' always have the same binding time"?

An obvious idea is to "tag" type variables with their origin, but this only gives a partial solution. Suppose for example a number of type variables are union-ed, and then the ECR is made dynamic. This makes *all* variables dynamic, and this can be showed by the analysis, but it says nothing about causes and effects.

5.9.2 The granularity of binding times

The analysis described in this chapter is flow-insensitive and a summary analysis. A variable is assigned one binding time throughout a function body.

The generating-extension principle seems intimately related to uniform binding time assignments. A variable cannot not "change". A poor man's approach to flow-sensitive binding time assignment would be by renaming of variables. Clearly, this can be automated. We have not investigated this further, but we suspect some gains are possible.

5.9.3 Struct variants

We have described a polyvariant analysis that allows context-sensitive analysis of functions. The parameters of a function are ascribed different binding times according to the call-site. Notice, however, that this is not the case for *structs*: a struct member only exist in one variant.

Extension of the analysis to accommodate variants of struct definitions is likely to improve binding time separation.

5.9.4 Analysis of heap allocated objects

The binding time assignment to heap-allocated objects are based on object's birth-place. All objects allocated from the same call-site is given the same binding time. A more fine-grained analysis seems desirable. Notice that the inter-procedural binding-time analysis improves upon the binding time separation of objects, since heap allocation in different variants can be given different binding times.

5.10 Conclusion

We have developed a constraint-based polyvariant binding-time analysis for the Ansi C programming language.

We specified *well-annotatedness* via non-standard type systems, and justified the definition with respect to the generating-extension transformation.

Next we gave a constraint-based formulation, and developed an efficient constraint-based analysis. An extension to polyvariant analysis based on static-call graphs was also described.

We have implemented the analysis in the *C-Mix* system, and provided some experimental results. As evident from the figures, the analysis is very fast in practice.

Chapter 6

Data-Flow Analysis

We develop a *side-effect analysis* and an *in-use analysis* for the C programming language. The aim of the side-effect analysis is to determine side-effecting functions and assignments. The in-use analysis approximates the set of variables (objects) truly used by a function.

The purpose of data-flow analysis is to gather static information about program without actually running them on the computer. Classical data-flow analyses include common subexpression elimination, constant propagation, live-variable analysis and definition/use analysis. The inferred information can be employed by an optimizing compiler to improve the performance of target programs, but is also valuable for program transformations such as the generating-extension transformations. The result of an analysis may even be used by other analyses. This is the case in this chapter, where explicit point-to information is employed to track pointers.

Compile-time analysis of C is complicated by the presence of pointers and functions. To overcome the problem with pointers we *factorize* the analysis into a separate pointer analysis and data flow analysis. Several applications of the pointer analysis developed in Chapter 4 can be found in this chapter.

The side-effect analysis approximates the set of unconditional and conditional side-effects in a function. A side-effect is called conditional if its execution is controlled by a test, *e.g.* an `if` statement. We show how control-dependence calculation can be used to determine conditional side-effects, and formulate the analysis as a monotone data flow framework. An iterative algorithm is presented.

The in-use analysis is similar to live-variable analysis, but deviates in a number of ways. It yields a more fine-grained classification of objects, and it is centered around functions rather than program points. For example, the in-use analysis may give as result that for a parameter of a pointer type, only the address is used, not the indirection. The analysis is specified in classical data flow framework.

Both analyses are employed in the generating-extension transformation to suspend conditional side-effects, and avoid specialization with respect to non-used data, respectively.

This chapter mainly uses techniques from classical data-flow analysis. We present, however, the analyses in a systematic and semantically founded way, and observe some intriguing similarities with constraint-based program analysis.

6.1 Introduction

Data-flow analysis aims at gathering information about programs at compile-time. In this chapter we consider two classical program analyses, *live-variable analysis* and *side-effect analysis*. These have been studied extensively in the Fortran community, but to a lesser extent for the C programming language. The main reason being the pointer concept supported by C. The solution we employ is to *factorize* the analyses into two parts: an explicit pointer analysis and a data flow analysis. Thus, this chapter also serves to give applications of the pointer analysis developed in Chapter 4, and illustrate its usefulness.

The analyses developed in this chapter have an application in partial evaluation. The side-effect analysis is employed to track down conditional side-effects which must be suspended (by the binding-time analysis), and the in-use analysis is used to prevent specialization with respect to unused data.

6.1.1 Data-flow analysis framework

Recall that a function is represented as a single-exit control-flow graph $G = \langle S, E, s, e\rangle$ where S is a set of statement nodes, E a set of control-flow edges, and s and e are unique start and exit nodes, respectively. A program is represented via an inter-procedural control-flow graph G^*.

A *monotone* data-flow analysis framework (MDFA) is a tuple $D = \langle G, L, F, M\rangle$ of an (inter-procedural) control-flow graph G, a semi-lattice L (with a meet operator), a monotone function space $F \subseteq \{f : L \to L\}$, and a propagation-function assignment M [Marlowe and Ryder 1990b]. The assignment M associates to all statements (basic blocks) a propagation function $f \in F$.[1] The framework is called *distributive* if the function space F is distributive.

Example 6.1 In *constant propagation analysis* the lattice is specified by '$\{\bot \sqsubset n \sqsubset \top\}$', $n \in I\!N$, and F consists of functions abstracting the usual operators on L. For example, '$\bot + 2 = 2$' and $4 + \top = \top$. Constant propagation is a monotone data flow problem, but it is not distributive. **End of Example**

An *optimal solution* at a program point[2] n to a data flow problem is defined as

$$\mathrm{MOP}(n) = \bigwedge_{\pi \in \mathrm{path}(n)} f_\pi(1_L),$$

where path(p) denotes the paths from s to n. This is called the *meet over all paths solution.* Implicit in the meet over all paths is the so-called "data-flow analysis assumption": all paths are executable.

A *maximal fixed-point solution* to an MDFA framework is a maximal fixed-point to the equations

$$\mathrm{MFP}(s) = 1, \quad \mathrm{MFP}(n) = \bigwedge_{n' \in \mathrm{pred}(n)} f_{n'}(\mathrm{MFP}(n'),$$

[1]We give a syntax-based approach and will omit M.

[2]We often identify a program point with a statement node.

where pred(n) is the set of predecessor nodes of n.

A fixed-point can be computed via standard iterative algorithms. It holds that MFP $\leq$ MOP if F is monotone, and MFP $=$ MOP if F is distributive [Kam and Ullman 1977]. Thus, in the case of distributive problems, the maximal fixed-point solution coincides with the optimal solution. On the other hand, there exists instances of MDFA such that MFP $<$ MOP [Kam and Ullman 1977].

6.1.2 Solutions methods

A solution to a MDFA can be found via *iterative* [Kildall 1973,Kam and Ullman 1976] or *elimination* [Allen and Cocke 1976,Graham and Wegman 1976,Ryder and Paull 1986] algorithms. Iterative algorithms propagate values through the data flow functions F to obtain a solution; elimination algorithms reduce the control-flow graphs and compose propagation functions accordingly, and then apply the composed function to the problem.

Theoretically, most elimination algorithms exhibit lower worst-case complexity than iterative algorithms, but on many problems they are equally fast [Kennedy 1976]. Further, elimination algorithms are not guaranteed to work on irreducible flow graphs for all kind of problems. In this chapter we shall only consider iterative methods.

A MDFA satisfying $\forall v \in L : f(v) \geq v \wedge f(1)$ is called *rapid* [Kam and Ullman 1976]. It can be shown that at most $d(G) + 3$ iterations is needed by an iterative algorithm to compute the MFP solution, where d is the loop connectedness of G (which essentially corresponds to the number of nested loops). Essentially, rapidness means that the contribution of a loop is independent of the (abstract) values at the loop entry. Since the loop nesting in most programs are modest, rapid data-flow problems are tractable and efficient in practice.

Example 6.2 Constant propagation is fast. Fastness means that one pass of a loops is sufficient determine its contribution [Marlowe and Ryder 1990b]. **End of Example**

6.1.3 Inter-procedural program analysis

Local analysis is concerned with analysis of basic blocks. Global analysis considers the data flow between basic blocks. Intra-procedural analysis focuses on analysis of a function body and makes worst-case assumption about function calls. Inter-procedural analysis is centered around the propagation of data through functions. The aim of inter-procedural analysis is to differentiate the contexts a function is called from, to avoid spurious information to be propagated.

Example 6.3 Consider constant folding analysis in the following program.

```
int main(void)            int foo(int n)
{                         {
   int x, y;                 return n + 1;
   x = foo(2);            }
   y = foo(3);
}
```

Inter-procedural analysis will merge the two calls and approximate the result of '`foo()`' by $\top$, since '`n`' gets bound to both 2 and 3. Inter-procedural analysis avoid interference between the two calls, and maps '`x`' to 3 and '`y`' to 4. **End of Example**

In Chapter 4 and Chapter 5 we have conducted inter-procedural context-sensitive analysis on the basis of a program's static-call graph. For example, for each context a function may be called from, separate point-to information is available.

6.1.4 Procedure cloning

Inter-procedural analysis is mainly concerned with the *propagation* of information through functions. Consider now the use of data-flow information *in* a function body.

Traditionally, optimizations based on inter-procedural analyses use a *summary* of all calls. Thus, all contexts the function appears in are merged. For example, constant folding in the program above (Example 6.3) will not give to any optimization: the summary of '`n`' is $\top$, since '`n`' is bound to both 2 and 3.

Suppose that '`foo()`' is copied, and the call-sites are changed accordingly. This enables constant folding: the expressions '`n+1`' can be replaced by 3 and 4, respectively. Explicit copying *before* program analysis is undesirable since it may increase program size exponentially.

Copying of functions *on the basis* of context-sensitive analyses is known as *procedure cloning* [Cooper *et al.* 1993,Hall 1991]. In the example above, a reasonable cloning strategy would create two versions of '`foo()`', but avoid copying if the second call was '`foo(2)`'.

Example 6.4 Procedure cloning of the program in Example 6.3

```
int main(void)         int foo1(int n)        int foo2(int n)
{                      {                      {
   int x, y;              /* n = 2 */            /* n = 3 */
   x = foo1(2);           return n + 1;          return n + 1;
   y = foo2(3);        }                      }
}
```

Constant folding may replace the expressions '`n + 1`' by constants. **End of Example**

Explicit procedure cloning seems a natural pre-transformation for the generating-extension transformation described in Chapter 3. We will therefore continue to assume that functions are copied according to a program's static-call graph before generating-extension transformation. Hence, a function exists in a number of variants, corresponding to the number of call contexts. Recall that the static-call graph maps a call and a variant number to a (called) function and a variant. The context dependent analyses of this chapter rely on context sensitive point-to information.

$$
\begin{array}{ll}
\mathcal{L}[\![\ c\ ]\!]\ i & = \{\} \\
\mathcal{L}[\![\ v\ ]\!]\ i & = \{v\} \\
\mathcal{L}[\![\ e_1.i\ ]\!]\ i & = \{S.i \mid\ o \in \mathcal{L}(e_1)\ i, \mathrm{TypOf}(o) = \langle \texttt{struct S}\rangle\} \\
\mathcal{L}[\![\ \texttt{*}e_1\ ]\!]\ i & = \bigcup_o \tilde{\mathcal{S}}(o,i) \qquad o \in \mathcal{L}(e_1)\ i \\
\mathcal{L}[\![\ e_1\texttt{[}e_2\texttt{]}\ ]\!]\ i & = \bigcup_o \tilde{\mathcal{S}}(o,i) \qquad o \in \mathcal{L}(e_1)\ i \\
\mathcal{L}[\![\ \mathit{otherwise}\ ]\!]\ i & = \{\}
\end{array}
$$

Figure 60: Computation of an expression's lvalues

6.1.5 Taming pointers

Consider live-variable analysis of an expression '`*p = *q`'. The assignment kills all variables the pointer '`p`' may point to, and uses all variables $*q$ may be aliased to. Without pointer information, worst-case assumptions must be made, *i.e.* both pointers can point to all objects. This degrades the accuracy of the analysis.

We will use the pointer analysis in Chapter 4 to approximate the usage of pointers. Recall that for every pointer the analysis computes a set of abstract locations the pointer *may* point to during program execution.[3] We shall assume the result of the analysis is available in the form of the map $\tilde{\mathcal{S}} : \mathrm{ALoc} \times \mathrm{Variant} \to \wp(\mathrm{ALoc})$, where the set of *abstract locations* ALoc was defined in Chapter 4. Further, recall that variant 0 is a summary variant describing the effect of all contexts a function is called from.

Definition 6.1 *Let $e \in \mathrm{Expr}$ be an expression. Let the set $\mathcal{L}(e)\ i$ approximating the* lvalues *of e in variant i be defined by Figure 60.* □

The definition of $\mathcal{L}$ is justified as follows.

A constant has no lvalue, and the location of a variable is denoted by its name.[4] The lvalue of a struct indexing is the lvalues of the corresponding field. In the case of a pointer dereference expression, the pointer abstraction is employed to determine the objects the subexpression may point to. Similarly for array index expressions. Other expressions have no lvalue.

Recall that the pointer analysis abstracts unknown and externally defined pointers by the unique abstract location 'Unknown'. For example, if $\mathcal{L}(e) = \{\mathrm{Unknown}\}$, it means that the lvalue of e is unknown.

6.1.6 Overview of chapter

The rest of this chapter is organized into three main section. Section 6.2 develops a *side-effect analysis*. We employ a standard algorithm for computation of control-dependence, and present the side-effect analysis as a monotone data-flow problem. Section 6.3 develops an *in-use analysis*. Both analyses use point-to information to approximate the usage of pointers. Section 6.4 list related work, Section 6.5 discusses further work and and conclude.

[3] "May" in the sense that it will definitely not point to an object not in the set.

[4] In practice, a name corresponds to a declarator, such that the locality of a variable can be checked.

6.2 Side-effect analysis

A function commits a side-effect when it assigns a non-local object. The aim of *side-effect analysis* is to determine side-effecting statements and functions. Accurate determination is undecidable due to pointers. The analysis computes a safe approximation of the set of statements and functions that *may side effect* at run time.

6.2.1 May side-effect

A *side-effect* occurs when a function executes an *object setting* operation that changes the value of a non-local object. Examples include '`g = 1`' (where '`g`' is a global variable), '`p->x = 1`' (where '`p`' points to a heap-allocated struct), '`f()`' (where '`f`' is a side-effecting function), '`scanf("%d", &g)`', and '`getch()`' (which side-effect the input buffer). The latter two examples illustrate that externally defined functions may side-effect.

Naturally, it requires examination of a function body to determine whether it commits side-effects. We will assume that externally defined functions are annotated *pure* if they do not commit any side-effect. The present analysis can be employed to derive '`pure`' annotations automatically.

Exact determination of side-effects is undecidable, even under the all-paths-executable assumption. The reason being the un-decidability of pointers [Landi 1992b]. Consider an assignment '`*p = 1`'. The classification depends on the pointer '`p`'. If '`p`' points to a non-local object, the expression is side-effecting. Since we only have imperfect pointer usage information in the form of may point-to sets, we shall approximate side-effects by *may side-effect.*

6.2.2 Side-effects and conditional side-effects

We differentiate between two kinds of side-effects: *conditional side-effects* and *unconditional side-effects.* In the following the latter is simply called a *side-effect.*

An assignment to a non-local object is called a *conditional side-effect* if the evaluation of the assignment is under control of a test, *e.g.* an `if` statement, appearing in the same function as the assignment. Thus, conditional side-effect is an intra-procedural property.

Definition 6.2 *Define the* side-effect domain $\mathcal{SE} = \{\bot, \dagger, \ddagger\}$ *with the following interpretation:*

$\bot$	*no side-effect*
$\dagger$	*non-conditional may side-effect*
$\ddagger$	*conditional may side-effect*

and the order $\bot \sqsubset \dagger \sqsubset \ddagger$*.* □

We use the elements of the side-effect domain to annotate assignments. The annotation $e_1 =^\dagger e_2$ denotes that the expression may side-effect (unconditionally), and $e_1 =^\ddagger e_2$ indicates that the expression may commit a conditional side-effect. The conditional side-effect annotation says nothing about the test that controls the side-effect. In practice it is convenient to annotate with control-dependences as well.

Example 6.5 Consider the annotated '`push()`' function below.

```
int stack[MAX_STACK], sp;
/* push: push v on stack */
void push‡(int v)
{
/* 1 */   sp +=† 1;
/* 2 */   if (sp < MAX_STACK)
/* 3 */      stack[sp] =‡ v;
/* 4 */   else
/* 5 */      fprintf(stderr, "Push: overflow\n");
}
```

The first assignment side-effects the global variable '`sp`'. The second assignment is under control of the `if`. **End of Example**

6.2.3 Using side-effect information

Recall that the generating-extension transformation in Chapter 3 expects all side-effects under dynamic control to be suspended by the binding-time analysis (Chapter 5). Without side-effect annotations the binding-time analysis must suspend all side-effects, and without pointer information the binding-time analysis must suspend all indirect assignments! A crude approximation to the set of conditional side-effects is the set of all side-effects. This is, however, too coarsely for practical usage. For instance, it would render impossible initialization of static, global data structures.

The result of the side-effect analysis can easily be employed to suspend side-effects under dynamic control. All assignments annotated by ‡ are candidates. If they depend on a dynamic test, they must be suspended.

Further, the analysis can be employed to derive '`pure`' annotations automatically. If a function contains no side-effects, it can be annotated '`pure`'.

6.2.4 Control dependence

A function is represented as a single-exit flow graph $G = (S, E, s, e)$, where S is a set of statement nodes, E a set of directed control-flow edges, and s and e unique start and end nodes, respectively. Intuitively, a node n is *control-dependent* on a node m if m is a branch node and n is contained in one the alternatives.

Definition 6.3 *Let $m, n \in S$. Node m is* post-dominated *by n, $m \neq n$, if every path from m to e contains n. Node n is* control-dependent *on m if i) there exists a non-trivial path π from m to n such that every node $m' \in \pi \setminus \{m, n\}$ is post-dominated by n, and ii) m is not post-dominated by n* [Zima and Chapman 1991]. □

Hence, for a node n to be control-dependent on m, m must have (at least) two exit edges, and there must be two paths that connect m with e such that one contains n and the other does not.

Control-dependence can easily be computed given a *post-dominator tree* as described by Algorithm 6.1. Post-dominator trees can be constructed by computation of dominators in the reverse control-flow graph [Aho *et al.* 1986].

Algorithm 6.1 *Computation of control-dependence for flow-graph $G = (S, E, s, e)$.*

1. *Construct the* post-dominator tree *T for G.*
2. *Define $E' = \{\langle m, n\rangle \in E \mid\ n$ not ancestor for m in $T\}$.*
3. *For all $\langle m, n\rangle \in E'$: traverse T backwards from n to m's parent node and mark all nodes n' as control-dependent on m.*

(*See Ferrante et al. for a proof of the algorithm* [Ferrante *et al.* 1987].) □

A post-dominator tree can be constructed in time $O(S)$ [Haral 1985] ($O(S\log(S))$ [Lengauer and Tarjan 1979]). An edge can be determined to be in the set E' in constant time, if the post-dominator tree is represented via bit vectors. Traversing the post-dominator tree T can be done in time $O(S)$ (the worst-case path length), hence the total marking time is $O(S^2)$.

Definition 6.4 *For a node $m \in S$, let $\mathcal{CD}(n)$ be the set of nodes on which n is control-dependent (and empty if n is control-dependent of no nodes).* □

Example 6.6 Consider the program in Example 6.5. Statement 3 is control-dependent on statement 2. Likewise, $\mathcal{C}(5) = \{2\}$. **End of Example**

Remark. Recall from Chapter 2 that we assume all conditional expressions are transformed into `if-else` statements. Therefore, a statement cannot be control-dependent on an expression.

Example 6.7 Consider binding-time analysis of the program fragment below. Control dependencies are indicated via program point.

```
/* 1 */ if ( e₁ )
/* 2 */    if ( e₂ )
/* 3 */       g = 2‡;
```

Statement 3 is correctly recorded to contain a conditional side-effect, and the control-dependence relation describes the dependency $\mathcal{CD}(3) = \{2\}$.[5]

Suspension of statement 3 depends on whether the test of one of the statements in the transitive closure of $\mathcal{CD}(3)$ contains a dynamic expression. **End of Example**

6.2.5 Conditional may side-effect analysis

We describe the context-insensitive conditional may side-effect analysis. The extension into context-sensitive analysis is straightforward (repeat analysis for each variant).

The *may-side effect analysis* is factorized into the following three parts:

1. Pointer analysis, to approximate point-to information (Chapter 4).

[5]Notice that 1 is *not* included in $\mathcal{CD}(3)$. On the other hand, $\mathcal{CD}(2) = \{1\}$.

2. Control-dependence analysis, to determine control-dependencies (Algorithm 6.1).

3. Conditional may-side effect approximation (Definition 6.5).

The conditional may side-effect analysis is defined as a monotone data-flow framework as follows.

Definition 6.5 Conditional may-side effect analysis *is given by* $D = \langle G^*, \mathcal{SE}, \mathcal{S}\rangle$*, where* $\mathcal{S}$ *is defined by Figure 61.* $\square$

The function CSE : $\mathcal{SE} \times \text{Node} \to \mathcal{SE}$ (Conditional Side-Effect) is defined by

$$\text{CSE}(s, n) = \mathcal{CD}(n) \neq \emptyset \wedge s = \dagger \to \ddagger \,[]\, s$$

that is, CSE returns '‡' if the statement contains a side-effect and is control-dependent on a statement. For a statement s we denote by $n_s \in S$ the corresponding statement node.

The equations in Figure 61 determines a map $\sigma : \text{Id} \to \mathcal{SE}$ from function identifiers to a side-effect anotation. A solution to the equations in Figure 61 maps a function f to ‡ if it may contain a conditional side-effect; to † if it may contain a (unconditional) side-effect, and to $\perp$ otherwise.

The rules for pre and post increment, and assignment use the function $\mathcal{L}$ to determine the lvalues of expressions. If it contains a non-local object, *i.e.* an object not locally allocated, the expression is side-effecting. To approximate the effect of indirect calls, point-to information is used.

The rules for statements checks whether a contained expression may commit a side-effect, and makes it conditional if the statement is control-dependent on a statement.

Lemma 6.1 *The analysis function* $\mathcal{S}$ *(for functions) defined in Figure 61 is* distributive *and* bounded*.*

Proof By inspection of the rules. $\square$

Lemma 6.2 *The analysis function* $\mathcal{S}$ *(for functions) defined in Figure 61 is* rapid*.*

Proof For all functions f, we must check $\forall y \in \mathcal{SE} : \mathcal{S}(f)[f \mapsto y] \sqsupseteq [f \mapsto y] \sqcup \mathcal{S}(f)[f \mapsto \perp]$, where environments are ordered point-wise. This is obvious by definition of $\mathcal{S}$. $\square$

Since the data-flow problem is distributive, a solution can be found by a standard iterative solving procedure [Aho *et al.* 1986,Kildall 1973].

Expressions

$$
\begin{array}{ll}
\mathcal{S}[\![\ c\ ]\!]\ L\ \sigma & = \bot \\
\mathcal{S}[\![\ v\ ]\!]\ L\ \sigma & = v \in L \to \bot \ \square\ \dagger \\
\mathcal{S}[\![\ e_1\texttt{.i}\ ]\!]\ L\ \sigma & = \mathcal{S}(e_1)\ L\ \sigma \\
\mathcal{S}[\![\ \texttt{*}e_1\ ]\!]\ L\ \sigma & = \mathcal{S}(e_1)\ L\ \sigma \\
\mathcal{S}[\![\ e_1\texttt{[}e_2\texttt{]}\ ]\!]\ L\ \sigma & = \bigsqcup_i \mathcal{S}(e_i)\ L\ \sigma \\
\mathcal{S}[\![\ \texttt{\&}e_1\ ]\!]\ L\ \sigma & = \mathcal{S}(e_1)\ L\ \sigma \\
\mathcal{S}[\![\ o\ e_1\ ]\!]\ L\ \sigma & = \mathcal{S}(e_1)\ L\ \sigma \\
\mathcal{S}[\![\ e_1\ o\ e_2\ ]\!]\ L\ \sigma & = \bigsqcup_i \mathcal{S}(e_i)\ L\ \sigma \\
\mathcal{S}[\![\ \texttt{alloc(}T\texttt{)}\ ]\!]\ L\ \sigma & = \bot \\
\mathcal{S}[\![\ ef\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}\ ]\!]\ L\ \sigma & = ef \text{ pure } \to \bigsqcup_i \mathcal{S}(e_i)\ L\ \sigma \ \square\ \dagger \\
\mathcal{S}[\![\ f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}\ ]\!]\ L\ \sigma & = \sigma(f) \sqcap \dagger \\
\mathcal{S}[\![\ e_0\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}\ ]\!]\ L\ \sigma & = \sigma(f) \sqcap \dagger \qquad f \in \mathcal{L}(e_0)\ 0 \\
\mathcal{S}[\![\ \texttt{++}e_1\ ]\!]\ L\ \sigma & = \mathcal{L}(e_1)\ 0 \not\subseteq L \to \dagger \ \square\ \mathcal{S}(e_1)\ L\ \sigma \\
\mathcal{S}[\![\ e_1\texttt{++}\ ]\!]\ L\ \sigma & = \mathcal{L}(e_1)\ 0 \not\subseteq L \to \dagger \ \square\ \mathcal{S}(e_1)\ L\ \sigma \\
\mathcal{S}[\![\ e_1\ aop\ e_2\ ]\!]\ L\ \sigma & = \mathcal{L}(e_1)\ 0 \not\subseteq L \to \dagger \ \square\ \bigsqcup_i \mathcal{S}(e_i)\ L\ \sigma \\
\mathcal{S}[\![\ e_1\texttt{,}\ e_2\ ]\!]\ L\ \sigma & = \bigsqcup_i \mathcal{S}(e_i)\ L\ \sigma \\
\mathcal{S}[\![\ \texttt{sizeof(}T\texttt{)}\ ]\!]\ L\ \sigma & = \bot \\
\mathcal{S}[\![\ \texttt{(}T\texttt{)}e_1\ ]\!]\ L\ \sigma & = \mathcal{S}(e_1)\ L\ \sigma
\end{array}
$$

Statements

$$
\begin{array}{ll}
\mathcal{S}[\![\ s \equiv e\ ]\!]\ L\ \sigma & = \text{CSE}(\mathcal{S}(e)\ L\ \sigma, n_s) \\
\mathcal{S}[\![\ s \equiv \texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2\ ]\!]\ L\ \sigma & = \text{CSE}(\mathcal{S}(e)\ L\ \sigma, n_s) \sqcup \bigsqcup_i \mathcal{S}(S_i)\ L\ \sigma \\
\mathcal{S}[\![\ s \equiv \texttt{switch (}e\texttt{)}\ S_1\ ]\!]\ L\ \sigma & = \text{CSE}(\mathcal{S}(e)\ L\ \sigma, n_s) \sqcup \mathcal{S}(S_1)\ L\ \sigma \\
\mathcal{S}[\![\ s \equiv \texttt{case}\ e\texttt{:}\ S_1\ ]\!]\ L\ \sigma & = \text{CSE}(\mathcal{S}(S_1)\ L\ \sigma, n_s) \\
\mathcal{S}[\![\ s \equiv \texttt{default:}\ S_1\ ]\!]\ L\ \sigma & = \text{CSE}(\mathcal{S}(S_1)\ L\ \sigma, n_s) \\
\mathcal{S}[\![\ s \equiv \texttt{while (}e\texttt{)}\ S_1\ ]\!]\ L\ \sigma & = \text{CSE}(\mathcal{S}(e)\ L\ \sigma, n_s) \sqcup \mathcal{S}(S_1)\ L\ \sigma \\
\mathcal{S}[\![\ s \equiv \texttt{do}\ S_1\ \texttt{while (}e\texttt{)}\ ]\!]\ L\ \sigma & = \text{CSE}(\mathcal{S}(e)\ L\ \sigma, n_s) \sqcup \mathcal{S}(S_1)\ L\ \sigma \\
\mathcal{S}[\![\ s \equiv \texttt{for (}e_1\texttt{;}e_2\texttt{;}e_3\texttt{)}\ S_1\ ]\!]\ L\ \sigma & = \text{CSE}(\bigsqcup_i \mathcal{S}(e_i)\ L\ \sigma, n_s) \sqcup \mathcal{S}(S_1)\ L\ \sigma \\
\mathcal{S}[\![\ s \equiv l\texttt{:}\ S_1\ ]\!]\ L\ \sigma & = \mathcal{S}(S_1)\ L\ \sigma \\
\mathcal{S}[\![\ s \equiv \texttt{goto}\ m\ ]\!]\ L\ \sigma & = \bot \\
\mathcal{S}[\![\ s \equiv \texttt{return}\ e\ ]\!]\ L\ \sigma & = \text{CSE}(\mathcal{S}(e)\ L\ \sigma, n_s) \\
\mathcal{S}[\![\ s \equiv \texttt{\{}\ S_1\texttt{;}\ldots\texttt{;}S_n\texttt{\}}\ ]\!]\ L\ \sigma & = \bigcup_i \mathcal{S}(S_i)\ L\ \sigma
\end{array}
$$

Functions

$$
\begin{array}{lll}
\mathcal{S}[\![\langle T, d_i, d_j, S_k\rangle]\!]\ \sigma = \ \sigma[f \mapsto \sigma(f) \sqcup S] & \text{where} & L = \text{LocalObjects}(d_i, d_j) \\
 & & S = \bigsqcup_k \mathcal{S}(S_k)\ L\ \sigma
\end{array}
$$

Figure 61: Side-effect analysis

6.2.6 Doing side-effect analysis

Algorithm 6.2 contains a simple iterative algorithm for conditional may side-effect analysis.

Algorithm 6.2 *Iterative side-effect analysis.*

```
$\sigma$ = [$f_i \mapsto \bot$];
do
   $\sigma_0$ = $\sigma$; $\sigma$ = $\mathcal{S}(f_i)\ \sigma$;
while ($\sigma \neq \sigma_0$)
```

□

Obviously, the above algorithm is not optimal. Better performance will be obtained by a work-list algorithm where functions are visited in depth-first order according to the call-graph [Horwitz *et al.* 1987].

Since annotation of statements solely depends on the side-effect of contained expressions, it can be done during the analysis.

6.3 Use analysis

An variable is *in-use* at a program point if its value is needed in an evaluation before it is redefined. This section develops an *in-use analysis.*

In-use analysis is similar to live-variable analysis [Aho *et al.* 1986] but differs with respect to back-propagation of liveness into functions. Furthermore, the analysis of this section is more fine-grained that classical live-variable analysis.

The analysis is applied in partial evaluation to avoid specialization with respect useless static values, *i.e.* values that are not used in the further computation.

6.3.1 Objects in-use

In-use information is assigned to all sequence points[6] and the entry and exit statement of a function. Intuitively, an object is *in-use* at a program point, if there exists a use of the object on a path to the exit node before it is redefined.

Definition 6.6 *Let p be a program point in a function $f = \langle S, E, s, e\rangle$. An object $o \in$* ALoc *is (locally)* in-use *at program point p if there exists a* use *of o on an intra-procedural path from p to e before o is assigned.*

For a function f, define $\mathcal{IU}(f) \subseteq$ ALoc *to be the set of objects in-use at the entry node s of f.* □

The notion of 'use' is made precise below. Intuitively, an object is used in an expression if its value is read from the store; this includes uses due to function calls. The set of *abstract locations* ALoc was defined in Chapter 4.

[6] For ease of presentation we will ignore that '&&' and '||' constitute sequence points.

Example 6.8 We have $\mathcal{IU}($`main`$) = \{a, a[]\}$ and $\mathcal{IU}($`inc_ptr`$) = \{ip\}$.

```
int main(void)            int *inc_ptr(int *ip)
{                         {
   int a[10], *p, *q;        return ip + 1;
   p = &a[0];             }
   q = int_ptr(p);
   return *q;
}
```

Notice that $ip[]$ is not in-use in function '`inc_ptr()`'.

If it was changed such that it dereferenced its parameter, we would get $\mathcal{IU}($`inc_ptr`$) = \{ip, a[]\}$, since '`ip`' points to the array '`a`'. **End of Example**

As opposed to classical live-variable analysis, in-use analysis expresses itself about objects, and not variables. For example, Classifying '`ip`' live (Example 6.8) means that both the pointer *and* the indirection is live.

6.3.2 In-use and liveness

The notions of in-use and liveness are similar but do not coincide. The main difference is that in-useness is not back-propagated into functions.

Example 6.9 Consider the following functions.

```
int global = 0;
int main(void)
{                        int foo(int x)
   int local;            {
   foo(local);              return x;
   return global;        }
}
```

Live-variable analysis classifies '`global`' as live throughout '`foo()`' due to the `return` statement in '`main()`'. In-use analysis reveals that only the parameter '`x`' is used by '`foo()`'. **End of Example**

In-use information is more appropriate for partial evaluation than live variables. Recall that program points (functions) need not be specialized with respect to dead values [Gomard and Jones 1991a]. For instance, by specialization to live variables, function '`foo()`' in Example 6.9 would be specialized with respect to '`global`' which is superfluous. On the other hand, in-use information is insufficient for *e.g.* register allocation or dead code elimination, which rely on liveness [Aho *et al.* 1986]. In languages without functions the notion of live variables and in-use variable is equal.

6.3.3 Using in-use in generating extensions

In-use information can be employed in generating extensions to avoid specialization with respect to useless values. A convenient way to convey in-use information is via bit-strings.

Since the number n of global variables is constant in all functions, we can employ an enumeration where the first n positions denote the in-useness of globals, and the following positions the in-useness of parameters and locals. The bit representation of in-use is defined inductively as follows. For an object of base type, '1' denotes in-use. For an object of pointer type, '1[B]' indicates that the pointer is in-use, and the B gives the in-useness of the indirection. Similarly for array types. In the case of a struct type, '1{B...B}' represents the in-use of the fields.

Example 6.10 The encoding of the in-use for '`inc_ptr()`' is `"1[0]"`. **End of Example**

Section 3.12.3 describes the usage of in-use information.

The set $\mathcal{IU}(f)$ for a function f contains the object in-use at entry of f. The computation of the indirections of a parameter of pointer type that are in-use can be done by the means of point-to information. If an object the pointer may point to is in-use, the indirection is in-use.

6.3.4 In-use analysis functions

We formulate the in-use analysis as a backward, monotone data-flow analysis over the set of abstract locations, with set union as meet operator. For each function f we seek a set $\mathcal{IU}_{fun}(f) \subseteq \text{ALoc}$ that describes the objects f uses.

In-use analysis of expressions is similar to live-variable analysis. For an expression e we have the backward data-flow equation

$$\mathcal{IU}_{before}(e) = \mathcal{U}(e) \cup (\mathcal{IU}_{exp}(e)_{after} \setminus \mathcal{D}(e))$$

where $\mathcal{U}$ is the object *used* by e, and $\mathcal{D}$ is the objects *defined* by e. Figure 62 contains the formal definition.

The definition of $\mathcal{U}$ and $\mathcal{D}$ is straightforward. The function $\mathcal{L}$ is employed to approximate the effect of pointers. In the case of function calls, the objects used are added. As apparent from the definition, we have given the context-insensitive version of the analysis.

In-use information valid before statement S is denoted by $\mathcal{IU}_{stmt}(S)_{before}$, and the corresponding information after S is written $\mathcal{IU}_{stmt}(S)_{after}$. The analysis functions are depicted in Figure 63.

The equations are straightforward. In the case of loops, information from the body and the entry is merged. In the case of a `return` statement, the imaginary "exit statement" S_{exit} is used instead of the "next" statement. This allows the general rule for statement sequences.

Definition 6.7 In-use analysis *is given by* $D = \langle G^*, \wp(\text{ALoc}), \mathcal{IU}_{fun}\rangle$*, where* $\mathcal{IU}_{fun}(f) = \mathcal{IU}_{stmt}(S_{entry})$ *for all functions* $f \in G^*$*, and* $\mathcal{IU}_{stmt}$ *is defined by Figure 63.* □

Use analysis function

$$\begin{array}{lcl}
\mathcal{U}[\![\ c\ ]\!] & = & \{\} \\
\mathcal{U}[\![\ v\ ]\!] & = & v \text{ is FunId?} \rightarrow \{\} \mathbin{[\,]} \{v\} \\
\mathcal{U}[\![\ e_1.i\ ]\!] & = & \mathcal{L}(e_1.i)\ 0 \cup \mathcal{U}(e_1) \\
\mathcal{U}[\![\ \texttt{*}e_1\ ]\!] & = & \mathcal{L}(*e_1)\ 0 \cup \mathcal{U}(e_1) \\
\mathcal{U}[\![\ e_1\texttt{[}e_2\texttt{]}\ ]\!] & = & \mathcal{L}(*e_1)\ 0 \cup \bigcup_i \mathcal{U}(e_i) \\
\mathcal{U}[\![\ \texttt{\&}e_1\ ]\!] & = & \mathcal{U}(e_1) \\
\mathcal{U}[\![\ o\ e_1\ ]\!] & = & \mathcal{U}(e_1) \\
\mathcal{U}[\![\ e_1\ o\ e_2\ ]\!] & = & \bigcup_i \mathcal{U}(e_i) \\
\mathcal{U}[\![\ \texttt{alloc(}T\texttt{)}\ ]\!] & = & \{\} \\
\mathcal{U}[\![\ ef\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}\ ]\!] & = & \bigcup_i \mathcal{U}(e_i) \\
\mathcal{U}[\![\ f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}\ ]\!] & = & \bigcup \mathcal{U}(e_i) \cup (\mathcal{IU}_{fun}(f) \cap (\text{LocalObject} \cup \text{GlobalObject})) \\
\mathcal{U}[\![\ e_0\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}\ ]\!] & = & \bigcup_i \mathcal{U}(e_i) \cup (\bigcup_{f \in \mathcal{L}(e_o)0} \mathcal{IU}_{fun}(f) \cap (\text{LocalObject} \cup \text{GlobalObject})) \\
\mathcal{U}[\![\ \texttt{++}e_1\ ]\!] & = & \mathcal{U}(e_1) \\
\mathcal{U}[\![\ e_1\texttt{++}\ ]\!] & = & \mathcal{U}(e_1) \\
\mathcal{U}[\![\ e_1\ aop\ e_2\ ]\!] & = & \bigcup \mathcal{U}(e_i) \\
\mathcal{U}[\![\ e_1\texttt{,}\ e_2\ ]\!] & = & \bigcup_i \mathcal{U}(e_i) \\
\mathcal{U}[\![\ \texttt{sizeof(}T\texttt{)}\ ]\!] & = & \{\} \\
\mathcal{U}[\![\ \texttt{(}T\texttt{)}e_1\ ]\!] & = & \mathcal{U}(e_1)
\end{array}$$

Define analysis function

$$\mathcal{D}_{exp}[\![\ e_1\ aop\ e_2\ ]\!] = \mathcal{L}(e_1)\ 0 \cup \mathcal{D}_{exp}(e_2)$$

In-use analysis for expression

$$\mathcal{IU}_{exp}(e)_{before} = \mathcal{U}(e) \cup (\mathcal{IU}_{exp}(e)_{after} \setminus \mathcal{D}_{exp}(e))$$

Figure 62: In-use analysis functions for expressions

The ordering on ALoc was defined in Chapter 4.

Lemma 6.3 *In-use analysis is a distributive data-flow analysis.*

Proof The analysis functions use union and intersect operators only. □

6.3.5 Doing in-use analysis

The in-use analysis is factorized into the two subcomponents

1. Pointer analysis (Chapter 4).
2. In-use analysis (Definition 6.7).

The distributivity implies that a solution to an in-use problem can be found via a standard iterative solving algorithm [Aho *et al.* 1986,Kildall 1973].

$$
\begin{array}{lcl}
\mathcal{IU}_{stmt}[\![\ e\ ]\!]_{before} & = & \mathcal{IU}_{exp}(e)_{before}, \\
 & & \mathcal{IU}_{exp}(e)_{after} = \mathcal{IU}_{stmt}(S)_{after} \\
\mathcal{IU}_{stmt}[\![\ \texttt{if}\ (e)\ S_1\ \texttt{else}\ S_2\ ]\!]_{before} & = & \mathcal{IU}_{ex}(e)_{before}, \\
 & & \mathcal{IU}_{stmt}(S_i)_{before} = \mathcal{IU}_{exp}(e)_{after}, \\
 & & \mathcal{IU}_{stmt}(S)_{after} = \bigcup \mathcal{IU}_{stmt}(S_i)_{after} \\
\mathcal{IU}_{stmt}[\![\ \texttt{switch}\ (e)\ S_1\ ]\!]_{before} & = & \mathcal{IU}_{exp}(e)_{before}, \\
 & & \mathcal{IU}_{stmt}(S_1)_{before} = \mathcal{IU}_{exp}(e)_{after}, \\
 & & \mathcal{IU}_{stmt}(S)_{after} = \mathcal{IU}_{stmt}(S_1)_{after} \\
\mathcal{IU}_{stmt}[\![\ \texttt{case}\ e\texttt{:}\ S_1\ ]\!]_{before} & = & \mathcal{IU}_{exp}(e)_{before}, \\
 & & \mathcal{IU}_{stmt}(S_1)_{before} = \mathcal{IU}_{exp}(e)_{after} \\
 & & \mathcal{IU}_{stmt}(S)_{after} = \mathcal{IU}_{stmt}(S_1)_{after} \\
\mathcal{IU}_{stmt}[\![\ \texttt{default:}\ S_1\ ]\!]_{before} & = & \mathcal{IU}_{stmt}(S_1)_{before}, \\
 & & \mathcal{IU}_{stmt}(S)_{after} = \mathcal{IU}_{stmt}(S_1)_{after} \\
\mathcal{IU}_{stmt}[\![\ \texttt{while}\ (e)\ S_1\ ]\!]_{before} & = & \mathcal{IU}_{exp}(e)_{before} \cup \mathcal{IU}_{stmt}(S_1)_{after}, \\
 & & \mathcal{IU}_{stmt}(S_1)_{before} = \mathcal{IU}_{exp}(e)_{after}, \\
 & & \mathcal{IU}_{stmt}(S)_{after} = \mathcal{IU}_{exp}(e)_{after} \\
\mathcal{IU}_{stmt}[\![\ \texttt{do}\ S_1\ \texttt{while}\ (e)\ ]\!]_{before} & = & \mathcal{IU}_{stmt}(S_1)_{before} \cup \mathcal{IU}_{exp}(e)_{after} \\
 & & \mathcal{IU}_{stmt}(S_1)_{after} = \mathcal{IU}_{exp}(e)_{before}, \\
 & & \mathcal{IU}_{stmt}(S)_{after} = \mathcal{IU}_{exp}(e)_{after} \\
\mathcal{IU}_{stmt}[\![\ \texttt{for}\ (e_1\texttt{;}e_2\texttt{;}e_3)\ S_1\ ]\!]_{before} & = & \mathcal{IU}_{exp}(e_1)_{before}, \\
 & & \mathcal{IU}_{exp}(e_2)_{before} = \mathcal{IU}_{exp}(e_1)_{after} \cup \mathcal{IU}_{exp}(e_3)_{after}, \\
 & & \mathcal{IU}_{stmt}(S_1)_{before} = \mathcal{IU}_{exp}(e_2)_{after}, \\
 & & \mathcal{IU}_{exp}(e_3)_{before} = \mathcal{IU}_{stmt}(S_1)_{after}, \\
 & & \mathcal{IU}_{stmt}(S)_{after} = \mathcal{IU}_{exp}(e_2)_{after} \\
\mathcal{IU}_{stmt}[\![\ l\texttt{:}\ S_1\ ]\!]_{before} & = & \mathcal{IU}_{stmt}(S_1)_{before}, \\
 & & \mathcal{IU}_{stmt}(S)_{after} = \mathcal{IU}_{stmt}(S_1)_{after} \\
\mathcal{IU}_{stmt}[\![\ \texttt{goto}\ m\ ]\!]_{before} & = & \mathcal{IU}_{stmt}(S_m)_{before}, \\
 & & \mathcal{IU}_{stmt}(S)_{after} = \{\} \\
\mathcal{IU}_{stmt}[\![\ \texttt{return}\ e\ ]\!]_{before} & = & \mathcal{IU}_{exp}(e)_{before}, \\
 & & \mathcal{IU}_{stmt}(S)_{after} = \{\}, \\
 & & \mathcal{IU}_{stmt}(S)_{exit} = \mathcal{IU}_{exp}(e)_{after} \\
\mathcal{IU}_{stmt}[\![\ \{\ S_i\ \}]\!]_{before} & = & \mathcal{IU}_{stmt}(S_1)_{before}, \\
 & & \mathcal{IU}_{stmt}(S_{i+1})_{before} = \mathcal{IU}_{stmt}(S_i)_{after}, \\
 & & \mathcal{IU}(S)_{after} = \mathcal{IU}_{stmt}(S_n)after
\end{array}
$$

Figure 63: In-use analysis functions for statements

Algorithm 6.3 *Iterative in-use analysis.*

```
for (f in G) {
  $\mathcal{IU}(f)$ = { };
  for (s in f) $\mathcal{IU}_{stmt}(s)_{before}$ = { };
}
while (!fixed-point)
   for (f in G) $\mathcal{IU}(f)$;
```

□

In an implementation, sets can be represented by bit vectors, enabling fast union and intersection operations. Since we only are interested in in-use information at branch statements and at entry to functions, the storage usage is modest.

6.3.6 An enhancement

A minor change can enhance the accuracy of the analysis. Consider the rules for function calls $\mathcal{IU}_{exp}(f(e_1, \ldots, e_n))$. Suppose that f defines a global variable o. This implies that o is not in-use before the call (unless it is part of the actual arguments). This can be taken into account by subtracting the set $\mathcal{D}_{fun}(f)$ of the *outward defined objects* from the in-use objects.

6.4 Related work

The data-flow analysis framework was first described by Kindall for distributive problems [Kildall 1973], and later extended to monotone propagation functions by Kam and Ullman [Kam and Ullman 1977]. We refer to Aho *et el.* for an introduction [Aho *et al.* 1986] and Marlowe and Ryder for a survey [Marlowe and Ryder 1990b].

6.4.1 Side-effect analysis

The most work in side-effect analysis of programs solves the more complicated problem of Def/Use [Aho *et al.* 1986]. Banning first factorized the problem into direct side-effects and induced side-effects due to aliasing [Banning 1979]. Cooper and Kennedy presented a linear time algorithm for the same problem [Cooper and Kennedy 1988].

Using an inter-procedural alias analysis, Landi *et al.* have developed a modification side-effect analysis for a subset of C. The analysis uses a factorization similar to Banning's analysis. Choi *et al.* have constructed an analogous analysis [Choi *et al.* 1993].

Neirynck has employed abstract interpretation to approximate side-effects in a function language [Neirynck 1988]. The problem is more complicated due to higher-order functions and closures, but the repertoire of pointer operations is limited.

6.4.2 Live-variable analysis

Live-variable analysis has been studied extensively in the literature, and used as an example problem for several data-flow framework [Aho *et al.* 1986]. Kildall presents an iterative algorithm for the classical, intra-procedural problem [Kildall 1973]. Kennedy presents a node-listing algorithm [Kennedy 1975]. A comparison of an iterative and an interval-based algorithms reveals that no method in general is more efficient than the other [Kennedy 1976].

Yi and Harrison have designed a live-interval analysis based on abstract interpretation for an imperative intermediate language [Yi and Harrison 1992]. It computes the interval of an object's birth-time and its dead.

6.4.3 Procedure cloning and specialization

Procedure cloning creates copies of functions for better exploitation of data flow information. This can also be seen as *specialization* with respect to *data flow* properties. Thus, there is an intimate connection between procedure cloning and function specialization.

6.5 Conclusion and Future work

We have formulated a side-effect and an in-use analysis as distributive data-flow analysis problems, and given iterative solving algorithms.

6.5.1 Further work

The analyses presented in this chapter have been formulated as context-insensitive. The extension to context-sensitive analyses is straightforward.

The in-use analysis has at the time of writing not been implemented. The side-effect analysis is (as formulated here) not optimal. It traverses the program's syntax tree repeatedly. A better approach is to derive the data-flow equations, and solve these by an efficient work-list algorithm. In this light, the analysis is similar to a constraint-based analysis.

6.5.2 Conclusion

The classical data-flow analysis framework is often criticized for being without firm semantically foundation. Even though we have not presented correctness proofs for the analyses nor a throughout specification in this chapter, it seems obvious that it feasible.

More interesting, the in-use analysis closely resembles a constraint-based analysis. For a statement S, let $\mathcal{IU}(S)_{before}$ denote a set-variable, and Figure 63 gives the corresponding constraint formulation of the problem. Thus, the difference between data-flow analysis and constraint-based analyses (in finite domains) is minor. It also shows that classical efficient data-flow algorithms may be of benefit in constraint-based program analysis.

Chapter 7

Separate Program Analysis and Specialization

Partial evaluation is a quickly evolving program specialization technique. The technology is now so developed and mature that it is being applied in realistic software engineering. In this chapter we study some of the problems that emerge when program transformation systems are applied to real-world programs.

A pervasive assumption in automatic, global program analyses and transformers is that a subject program consists of one module only. Existing partial evaluators are monolithic: they analyze and specialize whole programs. In practice, however, software is structured into modules each implementing different aspects of the complete system. Currently, the problem is often side-stepped by merging of modules into one big file, but this solution is infeasible and sometimes impossible. Storage usage can impose an upper limit on the size of programs.

In this chapter we investigate *separate analysis* and *separate specialization*. As an example problem we consider *separate binding-time analysis* that allows modules to be analyzed independently from other modules. This objective is twofold: it becomes possible to handle large programs in a convenient way, and modification of one module does not necessarily mean that all modules have to be analyzed from scratch again. This may for instance reduce the analysis overhead during the manual binding-time engineering often necessary to obtain good results.

Next we extend the framework and consider *incremental binding-time analysis* that accommodates modules to be changed without the binding-time solution having to be recomputed from scratch. Both the separate and the incremental analysis is based on constraint solving, and are extensions to the analysis developed in Chapter 5. Further, we sketch how the principles carry over to other analyses.

In the last part of the chapter we study *separate specialization*. Partial evaluation is an inherently global transformation that requires access to all parts of the program being transformed. We outline the problems and present some preliminary methods.

Separate analysis has not been implemented into the *C-Mix* system at the time of writing, but is expected to be of major practical importance.

7.1 Introduction

Program transformation by the means of partial evaluation is a quickly evolving program specialization technology which now has reached a state where it is being applied to non-trivial real-life problems. Traditional partial evaluators are, as other global transformers, *monolithic*: they analyze and transform whole programs. This conflicts with modern software engineering that advocates organization of software into cleanly separated modules.

Practical experiments with the *C-Mix* system have revealed that separate treatment of real-scale programs matters. So far, the problem has been overcome by merging different translation units into one big file. In practice, though, it is infeasible to analyze and to inspect binding time annotations in a 5,000 line program, say. Further, both the time and storage usage become critical. Some techniques for treatment of modules are obviously needed.

We have developed a *separate* and an *incremental* binding-time analysis. Further, we consider *separate specialization*. As example we use the binding-time analysis developed in Chapter 5, but the techniques carry over to other constraints-based analyses.

7.1.1 Partial evaluation and modules

Traditional partial evaluation is accomplished by the means of symbolic evaluation of the subject program. If '`mix`' is the partial evaluator and p the subject program, specialization with respect to some static input s is performed by $[\![\texttt{mix}]\!](\overline{p}^{pgm}, \overline{s}^{val}) \Rightarrow \overline{p_s}^{pgm}$, where p_s is the residual program.

In practice, p is separated into modules $p = m_1, \ldots, m_n$ where each module implements different aspects of the complete system. For example, one module opens and reads files, and another module carries out some computations. In very large systems it is infeasible to apply partial evaluation to all modules, and it is useless. Specialization of I/O modules is not likely to result in any significant speedup. Thus, in practice we want to specialize only some modules.

Applying '`mix`' to a module, $[\![\texttt{mix}]\!](\overline{m_i}^{pgm}, \overline{s}^{val})$ is not a solution. The module is incomplete and it may be non-trivial to deliver the static input to '`mix`'. For example, m_i might read a complicated data structure built by another module. It is not straightforward to construct the static input by "hacking" the other modules since '`mix`' uses its own representation of data. Further, it is an undesirable feature of a transformation system that programs have to be changed in order to be handled.

7.1.2 Modules and generating extensions

Consider now program specialization via generating extensions. To specialize module m_i we convert it into a generating extension $[\![\texttt{gegen}]\!](\overline{m_i}^{pgm}) \Rightarrow \overline{m_{gen}}^{pgm}$, and *link* the modules generating the static input. This is possible since the generating extension uses the same representation of static data as the subject program. Thus, the generating extension technique seems superior to traditional partial evaluation with respect to modules.

This is only a partial solution, however. If more than one module has to be specialized, both '`mix`' and '`gegen`' fall short. Suppose that m_i and m_j implement the code of interest for specialization. It is then likely that m_i, say, contains some external function calls to function defined in module m_j. Isolated analysis and specialization of m_i will *suspend* the calls to functions defined in m_j, which probably will prevent good results.

The problem both '`mix`' and '`gegen`' (more precisely, the binding-time analysis) faces is similar to the problem of type checking in the presence of modules. To check the types of functions in a module, a C compiler most rely on type declarations supplied by the user. Since modules contain no binding time annotations, a binding-time analysis most make worst-case assumptions: all external references are suspended. However, it is both undesirable and error-prone to indicate binding times manually. It is time consuming, and due to unnoticed phenomena they may be wrong. Binding times should be inferred automatically.

7.1.3 Pragmatics

So far users of partial evaluators have adopted a pragmatically oriented attitude and applied various "tricks" to overcome the problems. For example, by merging two files some externally defined functions may become defined. This does not, however, solve the basic problem, even when the merging is done by the system.

Partial evaluation is no panacea, and sometimes a user has to "binding-time engineer" a program to obtain good results. For instance, it may be necessary to suspend a static variable to avoid code explosion. In practice, inspection of a program's binding separation is often required. Examination of 5,000 lines code, say, is not an option for a programmer. It would be more convenient to inspect modules in separation.

Moreover, then a module is changed, the complete program must be analyzed from scratch again. Wasteful if the modification only affects a minor part of the program, possibly only the changed module.

By nature of binding-time analysis, it is not possible to analyze a module completely separately from the modules it uses. Binding-time analysis is an inherently global analysis that needs information about the status of externally defined identifiers. Moreover a change in one module may influence the binding time division in other modules.

A possible way would be to let the partial evaluator system maintain a *global binding-time solution* that is updated when modules are added, modified or removed. This way, only the relevant parts of a software systems have to parsed and analyzed due to a change, and modules do not have to be merged. The binding time division can be inspected module by module.

To make this possible, a *binding time signature* from each module must be extracted and given to a *global* binding time solver. A *binding-time signature* must capture the dependencies between the module and externally defined identifiers. The *global* analysis can then use the set of binding-time signatures to solve the global problem. Notice that a binding time signature only has to be computed every time a module is changed. Not every time a division has to be computed.

This chapter develops the needed techniques.

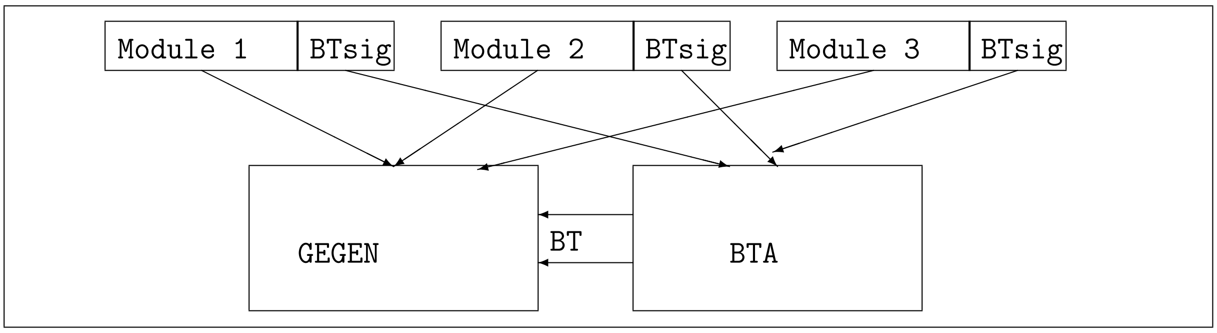

Figure 64: Separate binding-time analysis

7.1.4 Analysis in three steps

We develop a *separate* and an *incremental* version of the binding-time analysis from Chapter 5. Recall that the binding-time analysis is implemented via constraint-solving.

The new analysis proceeds in the three phases:

1. Each module is parsed and binding time information is extracted and reduced.
2. The global binding-time problem is initially solved via a traditional constraint solver.
3. The global solution is maintained via an incremental constraint solver to accommodate modifications of modules.

To represent and maintain the global binding time solution, a data base is employed. A user interface can extract information from the data base, *e.g.* to display annotated module code. Figure 64 illustrates the partial evaluation system.

7.1.5 Separate specialization

A motivation for separate compilation is memory limitation. Large programs may exhaust the compiler's symbol table, or build (too) big internal representations. The same problem is present in traditional partial evaluators, but less conspicuous in generating extensions. A generating extension uses the same representation of static data as the subject program.

However, other reasons are in favor of separate specialization. For example, a huge residual program may exhaust the compiler or give intolerably long compilation times.

7.1.6 Overview of chapter

The rest of the chapter is organized as follows. In Section 7.2 we describe the interference between modules and specialization. Section 7.3 extends previous work on constraint-based binding-time analysis into a *separate* analysis. Section 7.4 develops an *incremental constraint-solver* that implements incremental binding-time analysis. Problems related to separate specialization and transformation are discussed in Section 7.5. Related work is mentioned in Section 7.7, and finally Section 7.8 concludes and gives directions for further work.

7.2 The problem with modules

All but trivial programs are separated into modules. In the C programming language a module is a *translation unit*, which basically is a file of declarations. In this section we investigate the interaction between global[1] program analysis, specialization, and modules.

As an example we consider the following program which is built up of two translations units.[2]

```
/* File 1 */                           /* File 2 */
extern double pow(double,double);      #include <errno.h>
int errno;                             extern int errno;
int goal(int x)                        double pow(double n, double x)
{                                      {
   double d = pow(5.0,x);                 if (n > 0.0)
   if (!errno)                               return exp(x * log(n));
      printf("Result: %e\n,d);            errno = EDOM;
   return 0;                              return 0.0;
}                                      }
```

In file 1, a function '`pow()`' is declared and invoked from the '`goal()`' function. In file 2, the power function is defined. Errors are reported via the variable '`errno`', which for the sake of presentation is defined in the main module.

7.2.1 External identifiers

Suppose our aim is to specialize the main module with '`x`' dynamic. Since the first argument to the '`pow()`' is a (static) constant, we might expect the call to be specialized into a call '`pow_5()`'. However, since '`pow()`' is an externally defined function, it cannot be specialized — its definition is unknown at specialization time. Thus, nothing will be gained by specialization.

Consider now the binding times of the identifiers in file 2. Seemingly, the external variable '`errno`' is used in a static context, but since it is defined elsewhere, a binding-time analysis must necessarily classify it dynamic; external variables must to appear in residual programs.

In summary: all *references* to *externally defined identifiers* must be suspended. A consequence of this: all calls to the library functions, *e.g.* '`sin()`', are suspended. They are defined in the C library and '`extern`' declared in the include files, *e.g.* `<math.h>`.

7.2.2 Exported data structures and functions

Consider again file 1 defined above. Apparently, '`errno`' is a global variable belonging to file 1. However, in C are global identifiers by default *exported to other modules* unless explicitly defined to be local.[3] Hence, a global definition can in principle be accessed

[1] "Global" is here used in the meaning "whole program".

[2] We use translation unit and modules interchangeably.

[3] The scope of a global variable is restricted to a translation unit by the means of the 'static' storage specifier.

by all modules that declare it via '`extern`'. Looking at a module in isolation does not reveal whether other modules modify a global variable. For example in file 1 it cannot be determined that '`pow()`' in file 2 writes to '`errno`'.

Obviously, suspension of global identifiers is too conservative for almost all practical uses. In practice, it is convenient if the user via an option can specify whether a file makes up a program, such that global identifiers can be classified static.[4]

Functions are exported like global variables. This means that functions not explicitly made local by the means the '`static`' storage specifier potentially can be called from other modules. When modules are analyzed in isolation, call-sites and hence binding-time patterns, will not be known. To be safe, a binding-time analysis must suspend all arguments. Thus, no function specialization will take place. The situation can be alleviated slightly by copying of functions such that local calls get specialized.

In summary: all identifiers not specified '`static`' must be *suspended*.

7.2.3 Pure external functions

Suppose the call to '`pow()`' in file 1 was '`pow(5.0,2.0)`'. In this case we expect the call to be replaced by the result 32.0 provided the definition of '`pow()`' is available. That is, file 2 is linked to the generating extension of file 1. Naturally, for a binding-time analysis to classify a call to an external defined function static, it must "know" that the (compiled) definition eventually will become available at specialization time.

However, as noted above, a safe binding-time analysis without global program information must unconditionally suspend all calls to externally defined functions since they may side-effect variables. One way to improve on this is to provide the binding-time analysis with information about a function's side-effects. We define a function to be *pure* if it does not commit any side-effects during execution.

Assuming that externally defined function will be linked at specialization time, calls to pure function with static arguments can be classified static. In the example, '`pow()`' is not a pure function since it (may) side-effect the non-local variable '`errno`'. Actually, many C library functions report errors via the '`errno`' variable. Ways around this problem exists such that for instance the functions in `<math.h>` can be specified pure.[5]

In the *C-Mix* system, a specifier '`pure`' can be applied to specify side-effect free functions. For example, '`pure extern pow(double,double)`' would (erroneously) specify '`pow()`' to be pure.

7.3 Separate binding-time analysis

We consider the following scenario. A software system consists of a number of modules: '`file1.c`', ..., '`fileN.c`', and some standard libraries. The aim is to specialize some of the files, but not necessarily all. Using a monolithic binding-time analysis, all the relevant files would have to be merged and analyzed coherently.

[4]In *C-Mix* a user can by the means of an option specify that a file is a program.

[5]This is much trouble for almost no gaim since the majority of programmers do not even bother to check for errors.

In this section we describe a *separate binding-time analysis* that analyse modules separately. The idea is to perform the analysis in two steps: first essential binding-time information for each module is extracted and stored in a common data base. Next, the global binding-time problem is solved. The first step only has to be done one time for each module despite that the global problem is solved several times, *e.g.* due to modifications or manual binding-time engineering. Ideally, as must work as possible should be done in the first phase.

Some of the benefits are:

- Efficiency: a file only has to be parsed and analyzed once even though other files are modified repeatedly.

- Re-usability: a module can be used in several contexts but has only to be analyzed once. Prime example: library functions.

- Convenience: the user can analyze and inspect binding-time without having to change the logical structure of the program.

Moreover, the software does not have to be rewritten or rearranged to meet the requirements of the system, which is an issue when partial evaluation is applied to existing programs.

We refer to a static analysis that works across module boundaries as an *inter-modular analysis*, opposed to an *intra-modular analysis*. Binding-time analysis is inter-modular since the binding-times of one module (may) depend on the binding-times other modules, *e.g.* due to external variables. The output of a separate analysis shall equal the result of a intra-modular analysis of the union of all files.

Restriction: in this section we consider monovariant binding-time analysis only. To analyze languages like C, other inter-procedural and inter-modular analyses may be needed, for instance pointer and side-effect analysis. We return to this in Section 7.6.

7.3.1 Constraint-based binding-time analysis revisited

This section briefly reviews the binding-time analysis in *C-Mix*, see Chapter 5. The analysis is specified as a non-standard type-inference, and implemented by the means of constraint solving. In practice, the analysis is intermingled with parsing and static type inference but we give a self-contained presentation here.

The analysis basically consists of three steps:

1. Generation of a constraint system capturing the binding-time dependencies between expressions and subexpressions.

2. Normalization of the constraint system by exhaustive application of a set of rewrite rules.

3. Computation of a minimal solution to the normal-form constraint system.

The first step can be done by a syntax-directed traversal of the syntax tree. The normalization can be accomplished in time linear in the number of constraints.

Given a binding time classification of all variables (a division), it is easy to derive an annotation.[6]

Example 7.1 Consider the assignment '`errno = EDOM;`' from Section 7.2. Recall that '`errno`' is '`extern`' declared in file 2.

The following constraints could be generated:

$$\mathcal{C} = \left\{ \left\langle \begin{array}{c} \texttt{int} \\ \beta_{EDOM} \end{array} \right\rangle = \left\langle \begin{array}{c} \texttt{int} \\ S \end{array} \right\rangle, \left\langle \begin{array}{c} \texttt{int} \\ \beta_{errno=EDOM} \end{array} \right\rangle = \left\langle \begin{array}{c} \texttt{int} \\ \beta_{errno} \end{array} \right\rangle, \left\langle \begin{array}{c} \texttt{int} \\ \beta_{EDOM} \end{array} \right\rangle \preceq \left\langle \begin{array}{c} \texttt{int} \\ \beta_{errno} \end{array} \right\rangle \right\}$$

where β_{errno} denotes the binding time (variable) associated with '`errno`'. To the system the constraint $D \rhd \beta_{errno}$ would be added if file 2 was analyzed in isolation, effectively suspending '`errno`'. **End of Example**

A constraint system can be normalized by exhaustive application of a set of solution preserving rewrite rules. A normal form constraint system consists of constraints of the form $\left\langle \begin{array}{c} \tau_b \\ S \end{array} \right\rangle \preceq \left\langle \begin{array}{c} \tau_b \\ \beta \end{array} \right\rangle, \beta \rhd \beta'$.[7] A *solution* to a constraint system is a substitution from binding time variables to S and D such that all constraints are fulfilled. Since a minimal solution maps all β in $\left\langle \begin{array}{c} \tau_b \\ \beta \end{array} \right\rangle$ to S unless it is dynamic due to a dependency constraint $\beta_1 \rhd \beta$, it suffices to consider dependency constraints.

7.3.2 Inter-modular binding-time information

The binding time of a global identifier is determined by the module defining it and other modules' use of it, cf. Section 7.2. Without global information a binding-time analysis must revert to worst-case assumptions and suspend all global references. However, when it is known that an identifier eventually becomes defined, it may not be necessary to suspend it.

The set of identifiers exported by a module is called the *provided set*; the set if identifiers imported by a module is called the *required set* [Cooper *et al.* 1986b]. In the C language, global variables defined without use of the '`static`' storage specifier are in the provided set. Identifiers declared '`extern`' are in the required set.[8] Identifiers declared '`static`' are neither required nor provided; we often call those *private*.

Example 7.2 Consider file 2 in Section 7.2. The required set contains '`errno`'. The provided set includes '`pow`'. There are no private identifiers. **End of Example**

Even though a constraint system corresponding to a module can be normalized, in general a solution can not be found. There are two reasons: the binding times of required identifiers are unknown, and secondly, the use of provided identifiers is unknown.

[6]The division must also contain binding times for allocation calls and type definitions.

[7]See the corresponding theorem in Section 5.4.

[8]Assuming that extern declared identifiers are actually referenced.

The separate analysis proceeds in two main phases. The first phase *generates* and *normalizes* constraint systems for each module. This is a local (modular) phase; modules are analyzed independently. The second phase *solves* the global constraint system corresponding to all modules. This phase is global: it applies to all modules (even though the separate modules are not parsed and analyzed again).

Example 7.3 Consider the file 2 listed in Section 7.2. Phase one would generate a constraint system containing the constraints listed above. If the global constraint solver is applied, constraints to suspend external identifiers must be added. If the global constraint solver is applied to the constraint systems for both file 1 and file 2, all identifiers are defined, and no suspension constraints have to be added. **End of Example**

To keep track of symbols and constraint systems we use a global data base mapping identifiers to bt-types.[9] [Cooper *et al.* 1986a,Ross 1986].

7.3.3 Binding-time signatures

A module's *binding-time signature* consists of the following information:

- The binding-time type of all global identifiers.
- The provided and required sets.
- The binding time constraint system.[10]

The bt-types provide the link between an identifier and its binding-time variables. The provided and required sets can easily be identified via the '`extern`' and '`static`' storage specifiers. The (normalized) constraint system is a a set of equations over binding time variables.

Example 7.4 The binding-time signature of file 2 from Section 7.2 is listed below.

```
#file "file2.c"
extern errno: <int,T1>
       pow:   <(<double,T2>,<double,T3>),T4><double,T5>
#bta
T4     |> T5
 more constraints
```

First follows an identification of the translation unit. Next is listed the type of global identifiers and possibly storage specifiers. The last part contains the binding time constraint system.

We assume that binding time variables are unique, *e.g.* prefixed with the module's name. **End of Example**

[9]In practice the data base would distributed and kept with each file but this is of no importance.
[10]Add constraints/data-flow equations for other analyses.

```
Generate file.cmix containing binding-time information.
$ cmix -c file.c
Read file?.cmix, solve constraints, and generate file?-gen.c.
$ cmix -m file?.c
Compile the generating extension files
$ cc -c file?-gen.c gen.c
Link the object files to produce gen
$ cc file?-gen.o gen.o -lcmix -o gen
```

Figure 65: Separate generation of a generating extension

The number of binding time constraints are linear in the number of expressions, and typically much lower. In the case where all global identifiers are private, *i.e.* declared '`static`', all constraints can be solved. For convenience we add local variables to binding time signatures. This allows for example a user interface to enquire the data base about the binding time of variables in a convenient way.

7.3.4 Doing inter-modular binding-time analysis

The global solving process proceeds in three phases. First the relevant signature files are read, and a *symbol table* mapping identifiers to their type (and binding-time variables) is established. Identifiers declared in several modules are identified and the corresponding binding time variables unified. In practice this can done by adding equality constraints to the global constraint system. The symbol table is updated to reflect whether an identifier is defined or external for all modules. Static identifier are named uniquely such that no name conflicts occur. Finally, the global binding time constraint system is collected from the binding time signatures.

Next, the symbol table is scanned for remaining '`extern`' declared identifiers. Constraints suspending these are added. Further, constraints suspending an identifiers can be added, *e.g.* due to user annotations.

Lastly, the global constraint system is solved. The symbol table provides the link between an identifier and its binding time. The consumer of binding times, for instance the generating-extension transformation in Chapter 3, enquires the data base about identifiers binding time, see Figure 64.

Algorithm 7.1 *Inter-modular binding-time analysis.*

1. *Read binding-time signatures, build symbol table and set up global constraint system.*
2. *Add suspension constraints for externally defined identifiers.*
3. *Solve the global constraint system.*

The symbol table contains the computed division. □

We consider each phase below. In Figure 65 a "session" with a separate analysis is illustrated.

The symbol table

The symbol table provides the connection between an identifiers and its bt-types. We model it as a map $\mathcal{B} : \mathrm{Id} \to \mathrm{Storage} \times \mathrm{Type}$. In practice, the symbol table can be implemented via a hash table, say. The operations needed are 'lookup' and 'insert'.

During the scan of binding-time signatures, storage specifications are resolved. If a definition of a (previously) '`extern`' declared identifier is met, the '`extern`' flag is removed from the entry. Further, the bt-types of identical identifiers are unified, *e.g.* by addition of equality constraints to the global constraint system. This step corresponds to the actions taken by a linker to link separately compiled files.

Example 7.5 Consider once again the example program from Section 7.2. The symbol table is illustrated below.

$$\begin{array}{lll}
\texttt{errno} & \mapsto & \left\langle\begin{array}{c}\texttt{int}\\ \beta_{errno}\end{array}\right\rangle \\
\texttt{goal} & \mapsto & \left\langle\begin{array}{c}\left(\left\langle\begin{array}{c}\texttt{int}\\ \beta_{x}\end{array}\right\rangle\right)\\ \beta\end{array}\right\rangle \left\langle\begin{array}{c}\texttt{int}\\ \beta_{goal}\end{array}\right\rangle \\
\texttt{pow} & \mapsto & \left\langle\begin{array}{c}\left(\left\langle\begin{array}{c}\texttt{double}\\ \beta_{n}\end{array}\right\rangle, \left\langle\begin{array}{c}\texttt{double}\\ \beta_{x}\end{array}\right\rangle\right)\\ \beta\end{array}\right\rangle \left\langle\begin{array}{c}\texttt{double}\\ \beta_{pow}\end{array}\right\rangle
\end{array}$$

When the two files are linked, all identifiers become defined, as evident from lack of '`extern`' specifiers above. An extern variable, *e.g.* the io-buffer `struct _iobuf[]iob`[11] would appear in the symbol table as

$$\texttt{extern iob} \mapsto \left\langle\begin{array}{c}[\,]\\ \beta_1\end{array}\right\rangle \left\langle\begin{array}{c}\texttt{struct\_iobuf}\\ \beta_2\end{array}\right\rangle$$

Static identifiers are named uniquely, henceforth there is no need for a '`static`' storage specifier. **End of Example**

At the end of the analysis, the symbol table contains the computed division, for example, $\mathcal{B}(\texttt{errno}) = \left\langle\begin{array}{c}\texttt{int}\\ S\end{array}\right\rangle$.

Suspension constraints

The aim of this phase is to add constraints which suspend external identifiers. For an identifier x with bt-type T which is recorded to be external by the symbol table, a constraint $D \rhd T\#b$ is added to the constraint system.

Example 7.6 The constraint $D \rhd \beta_1$ suspends the '`iob`' variable. **End of Example**

It has been implicit in the above exposition that a *whole* program is binding time analyzed. It may, however, be convenient to apply the separate analysis to parts of a system. In this case constraints suspending global identifiers must be added to the global constraint system. We assume that a user option specifies whether the files constitutes a complete program.

[11]Declared in <stdio.h>.

Example 7.7 In experiments it is often useful to suspend static variables, *e.g.* to avoid code explosion or specialization of code with little prospective speedup. In *C-Mix*, a specifier '`residual`' can be employed for this. Constraints suspending '`residual`' declared identifiers can be added as part of a module's constraint system, or to the global system. The latter is convenient in an environment where the user interactively can inspect the effect of suspensions. **End of Example**

Constraint system solving

The global constraint system can be solved using the techniques developed in Chapter 5, that is, by normalization. Notice that although the constraint system for each module is normalized, the global constraint system needs not be in normal form. For example, an identifier may be assigned the result of an external, dynamic function. The runtime of the algorithm is linear in the number of constraints, see Section 5.5.

The solving procedure can implemented as described in Chapter 5 such that binding time variables in the symbol table are destructively updated with the solution.

7.3.5 Using binding-times

The approach described here deviates from the framework in Chapter 5 in that the syntax tree is not directly annotated by binding time as a result of the constraint solving. Instead, the binding times of identifiers are recorded in the symbol table.

As described in Section 5.4.7 it is easy to derive an annotation from a division by a simple bottom-up traversal of the syntax tree. This can either be done during the (generating-extension) transformation, or in a separate step.

7.3.6 Correctness of separate analysis

We formulate the correctness criterion for the separate binding-time analysis, and prove it correct. The correctness follows from the correctness of the monolithic analysis, Chapter 5.

Theorem 7.1 *Let '`file1.c`', ..., '`fileN.c`' be translation units. Applying the separate binding-time analysis to '`file1.c`', ..., '`fileN.c`' gives same result as the monolithic analysis applied to the union of the files.*

Proof The separate analysis resolves storage specifications the same way as when translation units are merged manually, *i.e.* it complies to the ANSI C Standard [ISO 1990]. This implies that the same suspension constraints are added to the global constraint system as the monolithic analysis would do. In the monolithic analysis, an identifier is assigned exactly one binding time variable. The equality constraints added in the separate analysis assures that identifiers with the same location are unified. Hence, the constraint system built by the separate analysis has the same solution as the system the monolithic analysis solves. □

7.4 Incremental binding-time analysis

The separate analysis of the previous section computes a program's division from scratch every time a module is changed. This section develops an *incremental analysis* that allows a solution to a constraint system to be updated according to changes in modules.

7.4.1 Why do incremental analysis?

Existing binding-time analyses are exhaustive; they re-compute the solution from scratch every time a part of the program changes. Clearly, even though modern analyses tend to be fast this may be a time-consuming task, and is inconvenient in an interactive programming environment where fast response time is essential. Using an incremental analysis, only the affected part of the solution has to be re-computed.

An example. Program specialization is still an engineering process: the user analyzes the functions, inspects the binding-times, and probably manually suspends a variable by insertion of a directive, *e.g.* a '`residual`' flag. From the analysis point of view, the change typically consists of the addition of a new constraint $D \rhd \beta$. Yet, an exhaustive analysis would generate the complete constraint system again and solve it. The separate binding-time analysis renders generating of the complete constraint system superfluous, but the solution of the global systems still have to be found from scratch.

Another example. Automatic binding-time annotation of programs during editing, for example in a structured programming environment, may provide the programmer with valuable information. The idea is that the programmer during the editing of a file interactively can see the binding times, and thus immediately avoids undesired program constructs. Since constraint-based binding-time analyses essentially generate constraints in a syntax-directed manner, the incremental analysis presented here can be used to maintain an evolving solution to a changing constraint set.[12]

In this section we solely consider binding-time analysis. Incremental pointer analysis is briefly considered in Section 7.6 and incremental versions of classical data-flow analyses are referenced in Section 7.7.

7.4.2 The basic idea

The basic idea in the incremental constraint solver is to maintain a solution while constraints are added or removed. In the original presentation of constraint-based binding-time analysis of an untyped lambda calculus, Henglein briefly described an extension accommodating addition of constraints [Henglein 1991]. His algorithm does not, however, supports deletion of constraints. Thus, a solution can only be made more dynamic, never the contrary. The reason for this is the use of destructively updated data structures.

The situation resembles the conditions for re-iteration of a fixed-point solver for finding a solution to an altered data-flow equation system [Ryder *et al.* 1988]. A necessary condition is that the solution to the new system is bigger than the old solution. This

[12]We do not claim that the algorithms developed in this chapter are sufficiently fast, though.

implies that if a user has inserted a '`residual`' specifier, it "cannot" be removed again, unless the solution is computed from scratch.

We describe an incremental constraint-solver which allows constraints to be both added and removed. The scenario is as follows. The global constraint solver maintains a solution for all the files currently in "scope", *e.g.* a program. When a file is updated, the old constraints are removed, and the new set of constraints added. When a constraint is removed, its effect on the solution is "undone".

7.4.3 The components of an incremental constraint solver

An incremental constraint solver has three parts. A map representing bindings of identifiers to their bt-type (symbol table),[13] and routines for adding and deleting constraints. We assume that constraint sets are pre-normalized such that all equality constraints are removed (via substitutions/unification).

Representation of the solution

A solution maps binding time variables to either S or D. If un-instantiated variables are interpreted as S, only bindings to D have to be represented. A binding-time variable β can be dynamic for two reasons: due to a constraint $D \rhd \beta$, or due to a constraint $\beta_1 \rhd \beta$ where β_1 is dynamic. Finally, notice that a variable may be dynamic due to more than one constraint. On the other hand, if a variable is dynamic due to exactly one constraint and that constraint is deleted, the solution changes such that the variable is static.

We represent this by a map $\mathcal{B} : \text{BType} \to \wp(BType) \times I\!N$. Suppose $\mathcal{B}(\beta) = (T, n)$. The first component $T \subseteq \wp(\text{BVar})$ is the (multi-)set[14] of variables which directly depends on β. A variable β_2 directly depends on β_1 if there exists a constraint $\beta_1 \rhd \beta_2$. For example, given constraints $\{\beta_1 \rhd^1 \beta_2, \beta_1 \rhd^2 \beta_2\}$ (where we have labeled constraints), we have that β_2 directly depends on β_1 due to the constraints 1 and 2. The number n denotes the number times β is "forced" to be dynamic. Thus, $n = 0$ means that β is static. Intuitively, n is the number of constraints $B \rhd \beta$ where B is D or a variable which (currently) is mapped to dynamic.

Example 7.8 Suppose the following constraint set is given by $\mathcal{C} = \{\beta_1 \rhd \beta_2, \beta_1 \rhd \beta_2, \beta_2 \rhd \beta_1, D \rhd \beta_3, D \rhd \beta_3\}$. The (current) solution is given by the map $\mathcal{B}$:

$$\mathcal{B} = [\beta_1 \mapsto (\{\beta_2, \beta_2\}, 0), \beta_2 \mapsto (\{\beta_1\}, 0), \beta_3 \mapsto (\{\}, 2)]$$

Variables β_1 and β_2 are static and depend on each other. The variable β_3 is dynamic, and no variables depends on it. Notice how the "dynamic-count" of β_3 is 2 since two dependency constraints force it to be dynamic. **End of Example**

The reason for the use of a count as opposed to a boolean flag for representing "dynamic" is the following. Suppose that a variable β is dynamic due to the constraints $D \rhd \beta$

[13]In practice, the symbol table also represents other information, *e.g.* storage specifiers.

[14]The same constraint may appear multiple times in the constraint set.

and $D \rhd \beta$. If the first constraint is removed, the solution does not change. However, if the second also is removed, the solution must be updated to reflect that β now is static. Thus, the count represents the "number of constraints that must be removed before β becomes static".

For simplicity we take $\mathcal{B}(\beta)$ undefined to mean $\mathcal{B}(\beta) = (\{\}, 0)$, *i.e.* β is static. In an implementation the binding map can be represented via a hash table '`hash`' with two methods: `lookup()` to search for a recorded binding, and `insert()` to destructively insert a new binding. We assume that the value D (dynamic) is pre-inserted into '`hash`' such that constraints $D \rhd \beta$ can be handled as other constraints.

Adding a constraint

We consider addition of a constraint while maintaining the current solution. Three cases are possible. The constraint does not effect the current solution (simply add it); the constraint makes a variable dynamic (update the binding map to reflect it); or the constraint introduces a new dependency between variables (update the binding map to reflect it). Since we assume D is represented as a variable, the first and last cases can be treated as one. Adding a constraint can by nature never make a solution less dynamic.

Algorithm 7.2 *Add set of constraints.*

```
/* Add constraints C' to the current constraint system C */
add_constraint(C')
{
   for (c = T1 ▷ T2 in C')
      add_dep(T1, T2);
   C ∪= C';
}
```

The function '`add_dep()`*' is defined by Algorithm 7.3 below.* □

The algorithm uses the auxiliary function '`add_dep()`'. A constraint of the form $D \rhd \beta$ forces β to become dynamic. This is implemented via the function '`dynamize`'. Other constraints simply causes the dependency lists to be updated.

Algorithm 7.3 *Dynamize and add dependency.*

```
/* Make T dynamic */
dynamize(T)
{
   (T0,n0) = hash.lookup(T);
   hash.insert(T,(T0,n0+1));
   if (n0 == 0) /* upd. dep. var */
      for (t in T0) dynamize(t);
}
```

```
/* Add dependency T1 ▷ T2 */
add_dep(T1, T2)
{
   (T0,n) = hash.lookup(T1);
   hash.insert(T1, (T0 ∪ { T2 }, n));
   if (n > 0) /* dynamic */
      dynamize(T2)
}
```

□

The function '`dynamize()`' increases the dynamic-count associated with a type variable. If the variable changes status from static to dynamic, all dependent variables are dynamized too. The function '`add_dep()`' adds a dependency between two type variables $\beta_1 \rhd \beta_2$. If β_1 is dynamic, the variable β_2 is dynamized.

Example 7.9 Suppose the current constraint system is as defined in Example 7.8. Add the constraints $\{D \rhd \beta_1, \beta_3 \rhd \beta_4\}$ using Algorithm 7.2. The new binding map is given by

$$\mathcal{B} = [\beta_1 \mapsto (\{\beta_2, \beta_2\}, 2), \beta_2 \mapsto (\{\beta_1\}, 2), \beta_3 \mapsto (\{\beta_4\}, 2), \beta_4 \mapsto (\{\}, 1)]$$

which is a correct solution to the current constraint system

$$\mathcal{C} = \{\beta_1 \rhd \beta_2, \beta_1 \rhd \beta_2, \beta_2 \rhd \beta_1, D \rhd \beta_3, D \rhd \beta_3, D \rhd \beta_1, \beta_3 \rhd \beta_4\}$$

as desired. **End of Example**

Deleting a constraint

Removing a constraint from the current constraint system may cause the solution to become "less" dynamic, *i.e.* some variables may change status from dynamic to static. This happens when the "last" constraint "forcing" a variable to be dynamic is removed.

Algorithm 7.4 *Remove set of constraints.*

```
/* Remove constraints C' from current constraint set C */
remove_constraint(C')
{
    for (c = T1 ▷ T2 in C')
       remove_dep(T1,T2); break; /* Remove dep */
    C \= C';
 }
```

The '`remove_dep()`' *function is defined by Algorithm 7.5.* □

Removal of a constraint $D \rhd \beta$ causes β to be "less" dynamic. This is implemented via the '`staticize()`' function. In other cases the dependency lists are updated. The set difference operator is a multi-set operator that removes one occurrence of each element in C' from C.

Algorithm 7.5 *Staticize and remove dependency.*

```
/* Make variable T more static */
staticize(T)
{
   (T0,n0) = hash.lookup(T);
   hash.insert(T,(T0,n0-1));
   if (n0 == 1)
      /* Staticize dependent variables */
      for (t in T) staticize(t);
}
```

```
/* Remove dependency T1 ▷ T2 */
remove_dep(T1,T2)
{
   (T0,n0) = hash.lookup(T1);
   hash.insert(T1,(T0\{T2},n0));
   if (n0 > 0)
      /* Staticize dependent variable */
      staticize(T2);
}
```

□

The algorithm '`staticize()`' decreases the dynamic-count of the variable. If it becomes static, that is, zero, dependent variables are staticized as well. The function '`remove_dep()`' removes a dependency $\beta_1 \rhd \beta_2$ between two variables. If β_1 is dynamic, β_2 is made less via '`staticize()`'.

Example 7.10 Suppose the constraints added in Example 7.9 are removed again.

Removing $D \rhd \beta_1$ causes the level of β_1 to be decreased by one, but it remains dynamic. Removing the constraint $\beta_3 \rhd \beta_4$ eliminates β_4 from the dependency set of β_3, and decreases the level of β_4 to zero.

Hence, modulo the empty (static) binding of β_4, we end up with the same solution as in Example 7.8, as expected. **End of Example**

Self-dependencies (as present in the examples above) can be handled the following way. Consider a constraint systems as a graph. When a constraint of the form $D \rhd \beta$ is removed from the constraint set, the strongly-connected component containing β is computed. If none of the variables in the component are forced to dynamic by D, all variables (if they are dynamic) are changed to be static.

7.4.4 Correctness of incremental binding-time analysis

This section proves the correctness of the incremental constraint solver. In the following $\mathcal{C}$ denotes the current constraint set and $\mathcal{B}$ the current solution.

Definition 7.1 *A map $\mathcal{B}$ is called a* valid solution *to a constraint set $\mathcal{C}$ if for all β in C, if $\mathcal{B}(\beta) = (\mathcal{T}, n)$, then:*

- $\mathcal{T} = \{\beta' \mid \exists(\beta \rhd \beta') \in \mathcal{C}\}$,
- $n = |\{D \rhd \beta \in \mathcal{C}\} \cup \{\beta' \rhd \beta \mid \mathcal{B}(\beta') = (\mathcal{T}', n'), n' > 0\}|$

and $\mathcal{B}(\beta) = (\{\}, 0)$ for all other variables. $\square$

Let a map $\mathcal{B}$ be given. The map obtained by application of Algorithms 7.2 and 7.4 on constraint system $\mathcal{C}'$ is denoted by $\mathcal{B}'$. Clearly, to prove the correctness of the incremental solver, it suffices to consider addition versus deletion of a single constraint.

Lemma 7.1 *Let $\mathcal{C}$ be a constraint set and c a constraint. Assume that $\mathcal{B}$ is a valid solution to $\mathcal{C}$. Then the map $\mathcal{B}'$ obtained by application of '`add_constraint()`' (Algorithm 7.2) yields a* valid solution *$\mathcal{B}'$ to $\mathcal{C}' = \mathcal{C} \cup \{c\}$.*

Proof (Informal) There are two cases to consider.

- $c = D \rhd \beta$. Let $(\mathcal{T}, n) = \mathcal{B}(\beta)$. If n is the number of constraints forcing β to be dynamic in $\mathcal{C}$, then $n+1$ is the number of constraints making β dynamic in $\mathcal{C}'$. If β is dynamic before the addition of c ($n > 0$) then $\mathcal{B}'$ is a valid solution for $\mathcal{C}'$ (holds also when β depends on itself). If β not is dynamic before the addition of c ($n = 0$), then all dependent variables correctly are updated to be dynamic in $\mathcal{B}'$.
- $c = \beta_1 \rhd \beta_2$. Let $(\mathcal{T}, n) = \mathcal{B}(\beta_1)$. If the variables in $\mathcal{T}$ depend on β_1, then the set $\mathcal{T} \cup \{\beta_2\}$ depend on β_1 after the inclusion of c. The binding time status of β_1 does not change due to the addition of c. If β_1 is dynamic ($n > 0$), then one more constraint makes β_2 dynamic. The correctness of this operation follows from the first part of the proof.

This shows the correctness of '`add_constraint()`'. $\square$

Lemma 7.2 *Let $\mathcal{C}$ be a constraint set and c a constraint. Assume that $\mathcal{B}$ is a valid solution to $\mathcal{C}$. Then the map $\mathcal{B}'$ obtained by application of '`remove_constraint()`' (Algorithm 7.4) is a* valid solution *to $\mathcal{C}' = \mathcal{C} \setminus \{c\}$.*

Proof (Informal) The proof is analog to the proof of Lemma 7.1. There are two cases to consider.

- $c = D \rhd \beta$. Let $(\mathcal{T}, n) = \mathcal{B}'(\beta)$. In the set $\mathcal{C}$ there is one less constraint that forces τ to be dynamic, hence $\mathcal{B}'(\beta) = (\mathcal{T}, n-1)$. If the deletion of c makes β static (*i.e.* $n = 1$), the dynamic-count of all dependent variables are decreased by one.
- $c = \beta_1 \rhd \beta_2$. Let $(\mathcal{T}, n) = \mathcal{B}(\beta_1)$. The dependency map in $\mathcal{B}'$ is correctly updated to reflect that β_2 does not dependent on β_1 due to c[15] If β_1 is dynamic ($n > 0$), then β_2 is dynamic due to one less constraint, namely c. The correctness of this follows from the first part of this proof.

This shows the correctness of '`remove_constraint()`'. $\square$

Theorem 7.2 *Let $\mathcal{C}$ be a constraint set and $\mathcal{B}$ a valid solution. The solution obtained by addition or deletion of a set of constraint by application of '`add_constraint()`' (Algorithm 7.2) and '`remove_constraint()`' (Algorithm 7.4) is a valid solution to the resulting constraint system.*

Proof Follows from Lemma 7.1 and 7.2. $\square$

This demonstrates the correctness of the incremental constraint solver.

7.4.5 Doing incremental binding-time analysis

An incremental update of the current binding-time division due to a modification of a file can be summarized as follows.

Algorithm 7.6 *Incremental binding-time analysis.*

1. *Read the file's signature file, and remove the constraints from the current constraint set. Possibly remove suspension constraints.*
2. *Perform the modification.*
3. *Apply the local part of the separate binding-time analysis to obtain a new constraint set.*

[15]Recall that more than one constraint $\beta_1 \rhd \beta_2$ may exist.

4. Read the signature files, and add the new constraints to the current constraint set.

5. Possibly add suspension constraints.

Step 2 and step 4 uses '`add_constraint()`' *and* '`remove_constraint()`'*, respectively.* □

Notice that during the removal of constraints, the global symbol table must be updated according to the changes. For example, if the definition of a variable if removed, the entry must be attributed with an '`extern`' storage specifier.

In step 4 it is crucial that the binding-time analysis uses the same variable names as previously. This is easily fulfilled by making variable names unique. In step 5, the symbol table must be scanned in order to add suspension constraints, *e.g.* due to externally defined identifier.

7.5 Separate specialization

We have described separate binding-time analysis, and developed an incremental constraint solver that accommodates modifications of modules. This section is concerned with *separate specialization*, that is, specialization of a program by individual specialization of the modules.

There are several convincing reasons for pursuing this goal. However, partial evaluation is an inherently global process that relies on the presence of the complete program. We describe these and outline necessary conditions for separate specialization. Finally, we consider specialization of library functions.

7.5.1 Motivation

Assume again the usual scenario where a program is made up by a set of modules `file1.c`,..., `fileN.c`. The aim is to specialize these. For this, we apply the separate binding-time analysis, transform each file (individually) into generating extensions, link *all* the generating extensions together, and run the executable. The idea is to specialize the files *separately*, that is, to run each generating extension `file-gen.c` separately. There are several advantages contained herein:

- A specialized (residual) program is often larger than its origin, and may thus exhaust a compiler. With separate specialization, the residual program is contained in several modules `file1-spec.c`,..., `fileN-spec.c` which are more likely to be manageable by a compiler.

- Manual binding-time engineering often has a rather limited impact on the overall specialization. Hence, changing one module normally causes only a few residual functions to be changed. There is no need to specialize the program from scratch again.

- Separate specialization enables re-use of residual code. For example, an often-used library can be specialized once to typical arguments, and shared across several specializations.

The procedure would then be: run each of the generating extensions `file1-gen.c`, ..., `fileN-gen.c` to obtain residual modules `file1-spec.c`, ..., `fileN-spec.c`, which then are compiled and linked to form the specialized program.

In the next section we analyze the problem and explain why separate specialization to some extent conflicts with partial evaluation.

7.5.2 Conflicts between global transformations and modules

We identify the following two main reasons why separate specialization fails: preservation of evaluation order and the passing of static values between module boundaries.

In imperative languages, dynamic functions can "return" static results by side-effecting non-local static variables. For practical reasons, it is desirable to allow this, for instance to initialize global data structures, or to heap-allocates partially static objects. When functions (possibly) have static side-effect, *depth-first* specialization of function calls are required. Otherwise the evaluation order may be changed. Since function calls may be across module boundaries, this renders separate specialization difficult.

Suppose we disallow non-local functions to have side-effects. This implies that a dynamic function call cannot affect the subsequent specialization of the caller, and thus the called function can be specialized at a later stage. However, this requires that values of static arguments are stored, and the value of global variables the function uses. This may be a non-trivial task in the case of values of composed types (*e.g.* pointers and structs) and heap-allocated data structures. Thus, separate specialization in general requires storing and establishing of computation states *between* program run.

7.5.3 Towards separate specialization

We say that a module is amenable to separate specialization provided all functions (that actually are called from other modules) fulfills:

- only have static parameters of base type,
- only uses non-local values of base type,
- accomplishes no side-effects.

The restrictions are justified as follows.

When a function only takes base type arguments and uses non-local values of base type, these can easily be stored in for example a file between module specializations. The last requirement assures that the evaluation order is preserved even though function specialization is performed *breadth-first* as opposed to the depth-first execution order.

More concretely, separate specialization can proceed as follows. When a call to a function defined in a module amenable for separate specialization is met, the call, the static arguments and the values of used global variables are recorded in a log (*e.g.* a file). A residual call is generated immediately without specialization of the function definition. After the specialization, the module is specialized according to the log.

Practical experiments are needed to evaluate the usefulness of this method. We suspect that the main problem is that functions amenable for separate specialization are mixed with other functions. Libraries, on the other hand, seem good candidates for separate specialization.

7.5.4 A example: specializing library-functions

Practical experiments have revealed that it sometimes is worth specializing standard-library functions such as '`pow()`'. For example, specialization of '`pow()`' to a fixed exponent speeds up the computation by a factor 2.[16]

As noted in Section 7.2, the most library functions are not pure, *e.g.* they may side-effect the error-variable '`errno`'. However, by defining the '`matherr()`' function, these functions can be made pure, and hence suitable for specialization.

The idea is to add generating extensions for library functions, and allow partially static calls to (some) externally defined functions. For example, suppose that a program contains a call '`pow(2.0,x)`' where '`x`' is a dynamic variable. Normally, the call would be suspended, but if a generating extension for '`pow()`' is available, the call (and the function) can be specialized.

Example 7.11 In *C-Mix*, a *generating math-lib* '`libm-gen`' is part of the system. The generating math-lib is linked to generating extensions as follows:

```
cc file-gen.c gen.c -lcmix -lm-gen
```

where `libcmix` is the *C-Mix* library. The binding-time analysis is informed about which functions that are defined in '`m-gen`'. **End of Example**

The speedup obtained by specialization of the ecological simulation software ERSEM, as described in Chapter 9, can to some extent be ascribed specialization of the power function. Notice that library functions only have to be binding-time analyzed and transformed once.

7.6 Separate and incremental data-flow analysis

In this section we briefly consider separate and incremental pointer and data-flow analysis. We have neither developed nor implemented separate or incremental pointer analysis.

7.6.1 Separate pointer analysis

Recall from Chapter 4 that the pointer analysis is a set-based analysis implemented via constraint solving. It uses inclusion constraints of the form $T_1 \subseteq T_2$. The techniques developed in Section 7.3 carries over to pointer analysis.

For each file, the constraint set is written into the module's signature file. The global solver reads in all signatures, sets up the global constraint system, and solves it using the

[16]On a Sun 4 using the Sun OS math library.

algorithm from Chapter 4. The result is a symbol table that maps each identifier to its abstraction.

Incremental pointer analysis can be developed following the lines of Section 7.4. In this case, the map must represent inclusions such that for a variable T, $\mathcal{B}(T) = \mathcal{T}$ where $\mathcal{T}$ is the set of variables T_1 where $T_1 \subseteq T$.

7.7 Related work

Separate compilation is a pervasive concept in most modern programming languages and a natural concept in almost all commercial compilers, but apparently little attention has been paid to separate analysis. For example, the the Gnu C compiler compiles function by function, and does not even perform any inter-procedural analyses [Stallman 1991].

Cooper *et al.* study the impact of inter-procedural analyses in modular languages [Cooper *et al.* 1986a]. Burke describes the use of a global data base to record inter-procedural facts about the modules being compiled [Burke 1993]. To our knowledge, there exists no partial evaluators which incorporate separate analysis nor specialization.

Type inference in the presence of modules is now a mature concept — at least in the case of functional languages. It therefore seems plausible that existing type-based binding-time analyses should be extend-able into separate analyses. Henglein gave some initially thought about this, but it has never been followed up nor implemented [Henglein 1991].

Consel and Jouvelot have studied separate binding-time analysis for a lambda calculus based on effect inference [Consel and Jouvelot 1993]. Since their used lambda calculus contains no global data structures or side-effects, their problem is somewhat simpler. An efficient implementation of their analysis based on constraint solving would probably look very like our algorithms.

Incremental analysis has mainly been studied in the framework of classical data-flow problems. For example reaching definitions [Marlowe and Ryder 1990a], available expressions [Pollock and Soffa 1989] and live-variable analysis [Zadeck 1984]. Ramalingam and Reps have developed an algorithm for incremental maintenance of dominator trees [Ramalingam and Reps 1994]. The analyses have not yet been extended to complex language featuring dynamic memory allocation, multi-level pointers or pointer arithmetic.

Freeman-Benson *et al.* have described an incremental constraint solver for linear constraints [Freeman *et al.* 1990]. Our constraint systems are simpler, and henceforth the maintaining of a solution is easier.

Separate compilation and re-compilation most closely resembles separate program specialization. Olsson and Whitehead describe a simple tool which allows automatic re-compilation of modular programs [Olsson and Whitehead 1989]. It is based on a global dependency analysis which examines all modules, and generates a Makefile. Hood *et al.* have developed a similar global interface analysis [Hood *et al.* 1986].

7.8 Further work and conclusion

In this chapter we have studied separate program analysis and specialization. Two algorithms for binding-time analysis of modules were developed. A *separate analysis* enables binding-time analysis of modules, *e.g.* to accommodate modifications, and an *incremental constraint solver* allows maintaining of a current solution without having to re-compute it from scratch. We also discussed separate specialization of modules, and identified the main problems.

7.8.1 Future work

The material in this chapter has not been implemented in the *C-Mix* system at the time of writing. Problems with separate pointer analysis and data-flow analysis remain to be solved.

The algorithms in this chapter do not support polyvariant analysis of programs. This complicates separate analysis of library functions, where polyvariant binding time assignment is critical. Extension to polyvariant analysis seems possible via an inter-modular call-graph. However, the techniques used in previous work does not immediately carry over. Recall that constraints over vectors were solved. The problem is that the length of the vectors are unknown until solve-time.

A main motivation for separate specialization is to reduce the memory usage during specialization. We have outlined a criteria for separate specialization of modules, but this seems too restrictive for practical purposes. More liberal conditions are left to future work.

7.8.2 Conclusion

We have completed the development on separate and incremental binding-time analysis, and presented algorithms and outlined a possible implementation. We expect separate analysis to have major practical importance. Program specialization still requires manual engineering of programs, and the possibility to examine modules instead of complete programs clearly is valuable.

The techniques so far are not sufficiently developed to accommodate separate specialization. In practice, residual programs easily become huge, and may exhaust the underlying compiler. We envision a *program specialization environment* which keeps track of functions to be specialized, and allows the technology to be applied when feasible and not when possible. In such an environment, the analyses developed in this chapter would be central.

Chapter 8

Speedup: Theory and Analysis

A partial evaluator is an automatic program optimization tool that has pragmatic success when it yields efficient residual programs, but it is no panacea. Sometimes specialization pays off well by the means of large speedups, other times it does not. In this chapter we study *speedup* in partial evaluation from both a theoretical and a practical point of view.

A program optimizer is said to accomplish *linear speedup* if the optimized program at most is a constant factor faster than the original program, for all input. It has for long been suspected and accepted that partial evaluation based on constant folding, reduction of dynamic expressions, specialization and unfolding, and transition compressing can do no better than linear speedup, there the constant factor can depend on the static input, but is independent of the dynamic input.

This gives an upper bound on the prospective speedup by specialization, *e.g.* there is no hope that specialization of an exponential-time algorithm yields a polynomial-time program. On the other hand, constants matter in practice. Even a modest speedup of 2 may have significance if the program runs for hours. We prove that partial evaluation cannot accomplish super-linear speedup.

Faced with a program, it is usually hard to predict the outcome of specialization — even after careful examination of the program's structure. We have developed a simple, but pragmatically successful *speedup analysis* that reports about prospective speedups, given a binding-time annotated program. The analysis computes a speedup interval such that the speedup obtained by specialization will belong to the interval.

The analysis works by computing *relative speedup* of loops, the objective being that most computation time is spent in loops. Due to its simplicity, the analysis is fast and hence feasible in practice. We present the analysis, prove its correctness, give some experimental results, discuss shortcomings, and introduce some improvements. Furthermore, we describe various applications of the analysis.

We also outline a technique that takes values of static variables into account. This enables more accurate estimates of speedups.

This chapter is an extended version of the paper [Andersen and Gomard 1992] which was joint work with Carsten Gomard. The chapter also owes to later work by Neil Jones [Jones *et al.* 1993, Chapter 6].

8.1 Introduction

During the last years partial evaluation has demonstrated its usefulness as an automatic program specialization technique in numerous experiments. In many cases, specialization has produced residual programs that are an order of magnitude faster. However, the same experiments have revealed that partial evaluation is no panacea: sometimes specialization gives little speedup, if any. A user without detailed knowledge of partial evaluation and the underlying principles is usually unable to predict the outcome of a specialization, without a time consuming manual inspection of residual programs.

In practice, the question "is specialization of this program worthwhile" most often be answered by applying the partial evaluator and running of the residual program. Our goal is to find a better way; in this chapter we study *speedups* in partial evaluation from two viewpoints: we prove a theoretical *limit* for prospective speedups, and describe an analysis that *predicts* speedups.

8.1.1 Prospective speedups

An informative answer to the above question is particularly desirable when partial evaluation is applied to computationally heavy problems. Clearly, it is wasteful to specialize a program and let it run for hours just to realize that nothing was gained. This is worse if the specialization itself is time consuming.

A "good" partial evaluator should ideally never *slow* down a program, but an upper limit for the obtainable speedup seems likely due to the rather simple transformation a partial evaluator performs. In particular, it does no "clever" transformations requiring meta-reasoning about subject programs, or spectacular change of algorithms, *e.g.* replaces a bubble-sort by a quick-sort. Other examples include replacing the naive recursive definition of the Fibonacci function by a linear iterative version.

An illuminating example is string matching where a pattern is searched for in a string. If m is the length of the pattern and n the length of the string, a naive string matcher possesses a runtime of $O(m \cdot n)$. For a fixed pattern, a Knuth, Morris and Pratt matcher [Knuth *et al.* 1977] runs in time $O(n)$. The catchy question is "can partial evaluation of a naive string matcher to a fixed pattern give a KMP matcher"? It turns out that this is not the case.

We prove that partial evaluation at most can accomplish *linear speedup.* This implies that partial evaluation of a program with respect to some static input at most gives a program that is a constant factor faster than the original subject program. This statement contains three parts.

First, the speedup obtained by specialization *depends* on the static input. We provide some examples that show the dependency. Secondly, the speedup is *independent* of dynamic input. Thirdly, super-linear speedup is impossible, that is, *e.g.* an exponential runtime cannot be reduced to a polynomial runtime.

We formulate and prove the Speedup Theorem in a general setting that captures many (imperative) programming languages.

8.1.2 Predicting speedups

An *estimate* of the prospective speedup, available before the specialization, would be valuable information. Then partial evaluation could be applied when *feasible* and not only when *possible* as is often the case. On the basis of a speedup estimate, a user could rewrite his program to improve the speedup, proceed with specialization if the speedup estimate is satisfactory, and otherwise simply forget about it! It is logical to combine speedup analysis with *binding-time debuggers* that allow inspection of binding times.

Another prospective is to let predicted speedup decide whether to perform an operation at specialization time or to suspend it. For example, unrolling of loops that contribute little speedup is undesirable due to increased code size, and should be avoided.[1]

Speedup estimation is clearly undecidable. We shall base our analysis on approximation of *relative speedup* and concentrate on loops, due to the fact that most computation time is spend in loops.

We have a simple, but pragmatically successful *speedup analysis* which approximates the speedup to be gained by an interval $[l, h]$. The interpretation is that for any static input, the residual program will be between l and h times faster than the source program. Since partial evaluation may result in infinite speedup, the upper bound may be infinity. Consider for example a program containing a completely static loop with a bound determined by the static input.

8.1.3 A reservation

To carry out analysis of speedups, we must make some simplifications. We shall assume that a basic operation, *e.g.* the addition of two numbers takes a fixed time. While this in general is true, it does not imply that the computation time of a "high-level" expression can be found by summing up the computation time of the expression's parts. Optimizing compilers may change expensive computation into cheaper instructions, and may even discard a useless expression.

We shall mainly ignore this aspect for the present. Practical experiments have shown that the speedup analysis gives reasonable, — and useful — information despite these simplifications. We return to an in-depth study of the interaction between specialization and classical code optimization in Chapter 9.

8.1.4 Overview of chapter

The remainder of the chapter is organized as follows. In Section 8.2 we define measurement of speedup and linear speedup, and prove that partial evaluation cannot accomplish super-linear speedup. Section 8.3 develops an automatic speedup analysis that computes a speedup interval on the basis of a binding-time annotated program. Section 8.4 describes an approach where the prospective speedup is computed during the execution of generating extensions. Section 8.5 cites related work, and Section 8.7 holds the conclusion and a list of further work.

[1]Technique: reclassify the loop as dynamic by the binding-time analysis or during specialization.

8.2 Partial evaluation and linear speedups

We consider a program represented as a control-flow graph $\langle S, E, s, e\rangle$, where S is a set of statement nodes, E a set of control-edges, and s, e unique start and exit nodes, respectively. Languages with functions, *e.g.* the C language, fits into this framework, see Chapter 2.[2] To make statements about speedups, we must relate run times of subject and specialized programs. This can be tricky since the subject program is a two-input program whereas the residual program only inputs the dynamic data.[3]

8.2.1 Measuring execution times

A *computation state* $\langle p, \mathcal{S}\rangle$ is a program point p and a store $\mathcal{S}$ mapping locations (variables) to values. If program execution passes control from a state $\langle p_i, \mathcal{S}_i\rangle$ to a new state $\langle p_{i+1}, \mathcal{S}_{i+1}\rangle$ it is written $\langle p_i, \mathcal{S}_i\rangle \to \langle p_{i+1}, \mathcal{S}_{i+1}\rangle$. A finite program execution is a sequence of transition steps

$$\langle p_i, \mathcal{S}_i\rangle \to \langle p_{i+1}, \mathcal{S}_{i+1}\rangle \to \cdots \to \langle p_j, \mathcal{S}_j\rangle$$

where $i < j$ and p_j is terminal. The program execution starts at the program point p_0 corresponding to s, and a store $\mathcal{S}_0$ initialized with the program input. Notice that a single statement or a basic block may constitute a program point — this is immaterial for the exposition.

Each transition has a cost in terms of runtime. For example, if program point p_i is an expression statement followed by a '`goto`' to program point p_k, the cost is the cost of the expression plus the control-flow jump. Thus, the runtime of a program applied to some input can be approximated by the length of the transition sequence leading from the initial program point to the final program point.

For a program p and input s, d we write $|[\![p]\!](s, d)|$ for the execution time. This means that for given program *and* input s and d, the *speedup* obtained by partial evaluation is given by

$$\frac{|[\![p]\!](s, d)|}{|[\![p_s]\!](d)|}$$

where p_s denotes p specialized with respect to d.

Example 8.1 Speedups are often given as the percentage improvement in execution time. In this thesis we solely state speedups in the form defined here. **End of Example**

For some input s let $|s|$ denote the *size* of s. We assume given an order on data such that for a sequence of data s_i, $|s_i| \to \infty$ is meaningful.

[2]In the following we assume 'control-flow graph' captures extended control-flow graphs.

[3]Without loss of generality we assume that subject programs take two inputs.

8.2.2 Linear speedup

Clearly, for a *fixed* static input s, the actual exetution time that occurs when running the specialized and subject programs depends on the dynamic input.

Definition 8.1 *Let p be a two-input program, and s a static input.*

1. *Define the* relative speedup *by*

$$\mathcal{SU}_s(d) = \frac{|[\![p]\!](s,d)|}{|[\![p_s]\!](d)|}$$

   for all dynamic input d.

2. *Define the* speedup bound *by*

$$\mathcal{SB}(s) = \lim_{|d|\to\infty} \mathcal{SU}_s(d)$$

   for all static input s.

If specialization of program p with respect to static input s loops, define the speedup $\mathcal{SU}_s(d) = \infty$ for all d. □

The speedup bound is normally taken to be the "speedup obtained by specialization".

The notion of *linear speedup* is defined in terms of the speedup bound [Jones 1989, Jones 1990,Jones *et al.* 1993].

Definition 8.2 *Partial evaluator '`mix`' accomplishes* linear speedup *on program p if the speedup bound $\mathcal{SB}(s)$ is finite for all s.*[4] □

Formulated differently, the definition requires that given a fixed static input s, there shall exist an $a_s \geq 1$ such that $a_s \cdot |[\![p_s]\!](d)| \leq |[\![p]\!](s,d)|$ for all but finitely many d.

Example 8.2 It is too strict to have $a_s \cdot |[\![p_s]\!](d)| < |[\![p]\!](s,d)|$. This would demand '`mix`' to optimize an arbitrary program which we by no means require. To see this, take s to be the "empty" input. **End of Example**

A largest speedup $\mathcal{SU}_s$ does not always exist. Consider for an example a program consisting of a single loop spending equally much time on static and dynamic computation, and add a static statement outside the loop. Any $a_s < 2$ bounds $\mathcal{SU}_s$ for all but finitely many d, but not $a_s = 2$. Still, 2 seems the correct choice for the speedup, since the static statement contributes little when the loop is iterated often enough.

8.2.3 Some examples

Some examples illustrate the relation between static input and speedups.

[4]It is here implicitly assumed that mix terminates.

No static data

Intuitively, when no static data is present, no speedup can be expected. This need not be true, however. If the program contains "static" data, *e.g.* in the form of statically defined arrays, specialization may give a speedup. Clearly, the speedup is independent from the dynamic input.

No dynamic data

Suppose that a program is specialized to *all* its input. The residual program then simply returns a constant. Obviously, the speedup is determined by the static input, and independent of the "dynamic" input.

Additive run times

Assume for a program p, that $|[\![p]\!](s,d)| = f(|s|) + g(|d|)$. The speedup by specialization of p is 1, since

$$\mathcal{SB}(s) = \lim_{|d|\to\infty} \mathcal{SU}_s(d) = \frac{f(|s|) + g(|d|)}{c + g(|d|)} = 1$$

cf. Definition 8.1.[5] This calculation relies on that the program eventually spends more time in dynamic computation that static computation. In practice, the static computation may dominate, so specialization is worthwhile.

String matching

In Chapter 10 we show how a naive string matcher can be specialized to yield an efficient KMP matcher by a slight change of the naive matcher. The speedup bound is given by $\mathcal{SB}(s) = |m|$, that is, the length of the pattern. Thus, the speedup is linear, but it depends heavily on the static input.

8.2.4 No super-linear speedup!

Jones posed the following as an open question [Jones 1989]: "If '`mix`' uses only the techniques program point specialization, constant folding, transition compression/unfolding, do there exist programs on which '`mix`' accomplishes super-linear speedups?". Equivalently, do there exists programs such that the speedup bound of Definition 8.1 is not finite.

Example 8.3 If '`mix`' makes use of *unsafe* reductions, such as '$e_1, e_2 \Rightarrow e_2$', introduces memorization, or eliminates common subexpressions, when it is easy to conceive examples of super-linear speedup. These reductions, however, may change the termination properties of the subject programs. **End of Example**

[5]We assume that $\lim g(d) = \infty$.

We prove that partial evaluation restricted to program point and function specialization, constant folding, and transition compression/unfolding cannot accomplish super-linear speedup. This is done by using the assumption that the partial evaluator terminates to place a bound on the speedup. Hence, we provide a negative answer to the question of Jones [Jones *et al.* 1993].

Without loss of generality we assume that every program point has a computation cost of 1, *i.e.* is a "basic" statement", and thus the run time of a program execution amounts to the length of the transition $\langle p_0, \mathcal{S}_0\rangle \to \cdots \to \langle p_n, \mathcal{S}_n\rangle$. Further, we assume an annotation classifying every statement as static or dynamic with respect to the initial division.

Theorem 8.1 (Speedup Theorem [Andersen and Gomard 1992,Jones *et al.* 1993]) *Suppose that '`mix`' is a safe partial evaluator, and p a program. Assume that '`mix`' terminates for all static input s. Then '`mix`' cannot accomplish super-linear speedup.*

Proof Let d be some dynamic input and consider a standard execution $[\![p]\!](s,d)$:

$$\langle p_0, \mathcal{S}_0\rangle \to \langle p_1, \mathcal{S}_1\rangle \to \cdots \to \langle p_n, \mathcal{S}_n\rangle$$

where we implicitly have assumed termination.[6] The initial store $\mathcal{S}$ contains both the static input s and the dynamic input d.

Each step in the computation involves computing the values $\mathcal{S}_{i+1}$ of the variables and a new program point, *e.g.* due to execution of an `if`. Suppose that all statements are "marked" with their binding times (even though we consider a standard execution). Consider what a partial evaluation would have done along the path $\langle p_0, \mathcal{S}_0\rangle \to \cdots \to \langle p_n, \mathcal{S}_n\rangle$.

Expressions marked static would be evaluated during specialization (= constant folding), and static control-flow statements would have been executed (= transition compression). In the case of a dynamic expression, some code would be generated (= reduction), and a residual jump would be generated in the case of a dynamic control-flow statement (= specialization). Beware, by specialization *more* than static evaluations/executions in the computation path above would in general be performed. Recall that we consider a standard execution and imagine *what* would have been done during specialization.

As usual, those computations done during specialization are called static, and those that are postponed to runtime are called dynamic. To calculate the speedup for this particular choice of s, d, simply sum up the costs of static and dynamic computation along the transition sequence above. If t_s is the cost of the static computations and similarly for t_d, we have

$$\mathcal{SU}_s(d) = \frac{t_s + t_d}{t_d}$$

where s, d are fixed.[7]

[6]If neither the original nor the residual program terminates the theorem is trivially fulfilled.

[7]We assume wlog. that a least one dynamic computation is carried out.

Assume that '`mix`' (the specializer or the generating extension) terminates in K steps. This means that if '`mix`' is applied to program p on static input s, there will be at most $K-1$ steps in the transition sequence above without any intervening code generation, since at each step '`mix`' either executes a statement, or generates code for a residual statement, and '`mix`' is no faster than standard execution.

Thus, for every dynamic statement, there are at most $K-1$ static statements, implying $t_s \leq (K-1) \cdot t_d$. This gives us

$$\mathcal{SU}_s(d) = \frac{t_s + t_d}{t_d} \leq \frac{(K-1)t_d + t_d}{t_d} \leq K$$

where K is independent of d. $\square$

Notice that '`mix`' always generates at least one residual statement; at least a '`return value`'. The estimate K above is far larger than the speedup that can be expected in practice, but the argument shows the speedup to be bounded.

8.3 Predicting speedups

An estimate of the prospective speedup, available before the residual program is run, would be valuable information. On the basis of a speedup approximation, a user can decide whether specialization is worthwhile or some binding-time improvements are necessary for a satisfactory result. This is especially important in the case of computational heavy problems as for example scientific computation [Berlin and Weise 1990].

Speedup estimation can be performed at three different stages, listed in order of available information:

1. By (time) analysis of the residual program; an extreme is to apply the original and the specialized programs to the dynamic input and measure their execution times.

2. By analysis of the subject program given its static input.

3. By analysis of the subject program given only a binding time division of the variables.

Clearly, the first approach gives the best results but also requires the most information. Further, it gives only one observed value of the speedup, and it is not obvious how to relate an empirical speedup with for instance binding time annotations. The second approach is advantageous over the third procedure since it can exploit the exact value of static variables. The third way is the most general due to the absence of both static and dynamic data, and thus the least precise. Nonetheless, useful results can be obtained, as we will see.

We present a *speedup analysis* based on the third approach, and study ways to implement the second. Input to the speedup analysis is a binding-time annotated program, and output is a *speedup interval* $[l, h]$, such that the actual speedup will be at least l and at most h, where h may be ∞.

8.3.1 Safety of speedup intervals

A *speedup interval* $I \subseteq \{x \in \mathbb{R} \mid x \geq 1\} \cup \{\infty\} = \mathbb{R}^\infty$ for a program p captures the possible speedup $\mathcal{SU}_s(d)$ for all s and d in a sense to be made precise below.[8]

A speedup interval $[l, h]$ is *safe* for p if the speedup $\mathcal{SU}_s(d)$ converges to an element in the interval when s, d both grow such that $|[\![p]\!](s, d)| \to \infty$. In general, the speedup will not converge to a fixed x as $|[\![p]\!](s, d)| \to \infty$, but we shall require that if program runs "long enough" it shall exhibit a speedup arbitrarily close to the interval.

Definition 8.3 (Safety of speedup interval) *A speedup interval* $[l, h]$ *is* safe *for* p *if for all sequences* s_i, d_i *where* $|[\![p]\!](s_i, d_i)| \to \infty$

$$\forall \varepsilon > 0 : \exists k : \forall j > k : \mathcal{SU}_{s_j}(d_j) = \frac{|[\![p]\!](s_j, d_j)|}{|[\![p_{s_j}]\!](d_j)|} \in [l - \varepsilon, h + \varepsilon]$$

□

Consider again the scenario from Section 8.2.2 where a loop contains equally much static and dynamic computation, and assume that the speedup is independent of the choice of s. Then a safe (and precise) speedup interval is [2,2].

8.3.2 Simple loops and relative speedup

We consider programs represented as control-flow graphs. For the present we do not make precise whether cycles, that is, loops in the program, include function calls. We return to this in Section 8.3.6 below.

A *loop* in a control-flow graph $\langle S, E, s, e \rangle$ is a sequence of nodes $n_i \in S$:

$$n_1 \to n_2 \to \cdots \to n_k, k \in \mathbb{N}$$

where each (n_i, n_{i+1}) is an edge and $n_1 = n_k$. A *simple loop* is a loop $n_1 \to \cdots \to n_k$, $k > 1$, where $n_i \neq n_j$ for all $1 \leq i < j \leq k$.

Let for each statement the *cost* $\mathcal{C}(n_i)$ be the *execution cost* of n_i. For example, $\mathcal{C}(n_i)$ can be the number of machine instructions implementing n_i, or the number of machine cycles necessary to execute n_i. We discuss the definition of the cost function in greater detail in Section 8.6.1. For notational convenience we define $\mathcal{C}_s(n_i)$ to be the cost $\mathcal{C}(n_i)$ of n_i if the statement is static and 0 otherwise, and similarly for $\mathcal{C}_d$. Thus, given a sequence of statements $n_1, \ldots, n_k$, the cost of the static statements is denoted by $\sum_{i \in [1,k]} \mathcal{C}_s(n_i)$.

Definition 8.4 (Relative Speedup in loop) *Let* $l = n_1 \to \cdots \to n_k$ *be a loop. The* relative speedup $\mathcal{SU}_{rel}(l)$ *of* l *is then defined by:*

$$\mathcal{SU}_{rel}(l) = \begin{cases} \frac{\mathcal{C}_s(l) + \mathcal{C}_d(l)}{\mathcal{C}_d(l)} & \textit{if } \mathcal{C}_d(l) \neq 0 \\ \infty & \textit{otherwise} \end{cases}$$

where $\mathcal{C}_s(l) = \sum_{i \in [1,k-1]} \mathcal{C}_s(n_i)$ *and* $\mathcal{C}_d(l) = \sum_{i \in [1,k-1]} \mathcal{C}_d(n_i)$. □

The relative speedup of a loop is a number in $\mathbb{R}^\infty$, and is independent of the values of variables.

[8]For addition involving ∞ we use $x + \infty = \infty + x = \infty$ for all $x \in \mathbb{R}^\infty$.

8.3.3 Doing speedup analysis

Given a program $\langle S, E, s, e\rangle$ is easy to find the set $\mathcal{L}$ of simple loops [Aho *et al.* 1986]. In practice it may be more convenient to compute the set of natural loops. The basic idea behind the analysis is that the relative speedup of a program p is determined by the relative speedup of loops when the program is run for sufficiently long time.

Algorithm 8.1 *Speedup analysis.*

1. *For all simple loops $l \in \mathcal{L}$ compute the* relative speedup $\mathcal{SU}_{rel}(l)$.
2. *The* relative speedup interval *is* $[\min \mathcal{SU}_{rel}(l), \max \mathcal{SU}_{rel}(l)]$, $l \in \mathcal{C}$.

$\square$

The speedup analysis does not take basic blocks (statements) outside loops into account. Clearly, the speedup of the loops will dominate the speedup of the whole program provided the execution time is large. However, the analysis can easily be modified to handle the remaining basic blocks by accumulating relative speedups for all paths through the program without loops. Without the revision, the analysis will have nothing meaningful to say about programs without loops.

Example 8.4 Consider the following contrived program which implements addition.

```
int add(int m, int n)
{
    int sum;
    /* 1 */ sum = n
    /* 2 */ while (m) {
    /* 3 */   sum += 1;
    /* 4 */   m -= 1;
    /* 5 */ }
    /* 6 */ return sum;
}
```

The basic blocks are: $\{1\}, \{2, 3, 4, 5\}, \{6\}$ where the second constitute a (simple) loop. Suppose that m is static but n dynamic. Then the statements 2, 4 and 5 are static and the rest dynamic. Letting the cost of statements be 1 "unit", we have the relative speedup of the loops is 4. The relative speedup interval is $[4, 4]$. **End of Example**

To see that the speedups for all non-simple loops are in the computed speedup interval, let us see that if $[u, v]$ is safe for loops l_1 and l_2 then it is also safe for a loop l composed from l_1 and l_2.

Lemma 8.1 *Let l_1 and l_2 be simple loops in a program p with a common start node, and assume that $[u, v]$ is safe for both l_1 and l_2. Then $[u, v]$ is safe for the loop l of any number of repetitions from l_1 and l_2.*

Proof Assume wlog. that $\mathcal{SU}_{rel}(l_1) \neq \infty$ and $\mathcal{SU}_{rel}(l_2) \neq \infty$, and l_1 executes m times and l_2 executes n times.

$$\begin{array}{rcl}
\mathcal{SU}_{rel}(l) & = & \dfrac{\mathcal{C}_s(l) + \mathcal{C}_d(l)}{\mathcal{C}_d(l)} \\
& = & \dfrac{m\mathcal{C}_s(l_1) + n\mathcal{C}_s(l_2) + m\mathcal{C}_d(l_1) + n\mathcal{C}_d(l_2)}{m\mathcal{C}_d(l_1) + n\mathcal{C}_d(l_2)} \\
& = & \dfrac{(\frac{m\mathcal{C}_s(l_1)+m\mathcal{C}_d(l_1)}{m\mathcal{C}_d(l_1)})}{(\frac{m\mathcal{C}_d(l_1)+n\mathcal{C}_d(l_2)}{m\mathcal{C}_d(l_1)})} + \dfrac{(\frac{n\mathcal{C}_s(l_2)+n\mathcal{C}_d(l_2)}{n\mathcal{C}_d(l_2)})}{(\frac{m\mathcal{C}_d(l_1)+n\mathcal{C}_d(l_2)}{n\mathcal{C}_d(l_2)})} \\
& = & \dfrac{\mathcal{SU}_{rel}(l_1)}{(\frac{m\mathcal{C}_d(l_1)+n\mathcal{C}_d(l_2)}{m\mathcal{C}_d(l_1)})} + \dfrac{\mathcal{SU}_{rel}(l_2)}{(\frac{m\mathcal{C}_d(l_1)+n\mathcal{C}_d(l_2)}{n\mathcal{C}_d(l_2)})}
\end{array}$$

Suppose $\mathcal{SU}_{rel}(l_1) \leq \mathcal{SU}_{rel}(l_2)$:

$$\begin{array}{rcl}
\mathcal{SU}_{rel}(l) & = & \frac{\mathcal{SU}_{rel}(l_1)}{(\frac{m\mathcal{C}_d(l_1)+n\mathcal{C}_d(l_2)}{m\mathcal{C}_d(l_1)})} + \frac{\mathcal{SU}_{rel}(l_2)}{(\frac{m\mathcal{C}_d(l_1)+n\mathcal{C}_d(l_2)}{n\mathcal{C}_d(l_2)})} \\
& \geq & \frac{\mathcal{SU}_{rel}(l_1)}{(\frac{m\mathcal{C}_d(l_1)+m\mathcal{C}_d(l_2)}{m\mathcal{C}_d(l_1)})} + \frac{\mathcal{SU}_{rel}(l_1)}{(\frac{m\mathcal{C}_d(l_1)+n\mathcal{C}_d(l_2)}{n\mathcal{C}_d(l_2)})} \\
& = & \mathcal{SU}_{rel}(l_1)
\end{array}$$

and similarly for $\mathcal{SU}_{rel}(l_2)$. Thus, we have

$$\mathcal{SU}_{rel}(l_1) \leq \mathcal{SU}_{rel}(l) \leq \mathcal{SU}_{rel}(l_2)$$

and therefore $[u, v]$ is safe for l. □

Theorem 8.2 (Safety of speedup analysis) *Assume the analysis computes a speedup interval* $[u, v]$ *for program* p. *Then* $[u, v]$ *is* safe *for* p.

Proof An upper bound $v = \infty$ is trivially safe, so assume $v \neq \infty$.

Consider the sequence of nodes n_i visited during a terminating computation $[\![p]\!](s, d)$ arising from the application of p to data s,d:

$$N = n_1 \rightarrow n_2 \rightarrow \cdots \rightarrow n_k$$

where a node n_i may occur several times in N.

To *delete* a simple loop $n_i \rightarrow \cdots \rightarrow n_j$, $i > j$ from N is to replace N by:

$$n_1 \rightarrow \cdots \rightarrow n_{i-1} \rightarrow n_{j+1} \rightarrow \cdots \rightarrow n_k$$

Delete as many simple loops as possible from N, and denote the set of remaining loops by $\mathcal{L}$. By definition, the remaining nodes in N occur only once, and the size of the program $|N|$ provides a bound on the number of non-loop nodes. Denote this set by $\mathcal{NL}$.

For the given program p and data s, d we calculate the speedup:

$$SU = \frac{\mathcal{C}_s(\mathcal{L}) + \mathcal{C}_s(\mathcal{NL}) + \mathcal{C}_d(\mathcal{L}) + \mathcal{C}_d(\mathcal{NL})}{\mathcal{C}_d(\mathcal{L}) + \mathcal{C}_d(\mathcal{NL})}$$

which can be rewritten to

$$SU = \frac{\frac{\mathcal{C}_s(\mathcal{L})+\mathcal{C}_d(\mathcal{L})}{\mathcal{C}_d(\mathcal{L})}}{\frac{\mathcal{C}_d(\mathcal{L})+\mathcal{C}_d(\mathcal{NL})}{\mathcal{C}_d(\mathcal{L})}} + \frac{\frac{\mathcal{C}_s(\mathcal{NL})+\mathcal{C}_d(\mathcal{NL})}{\mathcal{C}_d(\mathcal{NL})}}{\frac{\mathcal{C}_d(\mathcal{L})+\mathcal{C}_d(\mathcal{NL})}{\mathcal{C}_d(\mathcal{NL})}}$$

Now we will argue that for all $\varepsilon > 0$ there exists a K such that $\mathcal{SU}_s(d) \in [u - \varepsilon, v + \varepsilon]$ for if $|[\![p]\!](s, d)| > K$. For some s choose a sequence s, d_i such that $|[\![p]\!](s, d_i)| \to \infty$.

To the right of the $+$ we have that the numerator $\frac{\mathcal{C}_s(\mathcal{NL})+\mathcal{C}_d(\mathcal{NL})}{\mathcal{C}_d(\mathcal{NL})}$ is uniformly bounded, and that the denumerator converges to ∞ since $\mathcal{C}_d(\mathcal{L}) \to \infty$.

To the left of the $+$ we have that the denominator converges to 1. Thus, we conclude

$$SU \to \frac{\mathcal{C}_s(\mathcal{L}) + \mathcal{C}_d(\mathcal{L})}{\mathcal{C}_d(\mathcal{L})}$$

when $|[\![p]\!](s, d_i)| \to \infty$.

Since $\mathcal{L}$ is a multi-set of simple loops, we conclude that $[u, v]$ is safe for p, using Lemma 8.1. $\square$

Notice that the choice of a sequence s, d_i for which $|[\![p]\!](s, d_i)| \to \infty$ rules out completely static loops, *i.e.* specialization of the program is assumed to terminate.

8.3.4 Experiments

We have implemented the speedup analysis and examined its behavior on a number of examples. The implemented version solely considers loops, and has a differentiated cost function for statements and expressions. The analysis is fast; there is no significant analysis time for the examples here. All experiments have been conducted on a Sun SparcStation I, and times measured via the UNIX '`time`' command (user seconds). The programs '`polish-int`' and '`scanner`' originate from Pagan's book [Pagan 1990].

The '`add`' program is listed in Example 8.4. The program '`polish-int`' implements an interpreter for a simple post-fix language. In this example, the static input was a specification which computes the first n primes. The dynamic input was $n = 500$. The program '`scanner`' is a general lexical analysis that inputs a token specification and a stream of characters. It was specialized to a specification of 8 different tokens which appeared 30,000 times in the input stream.

Example	Run-time		Speedup	
	Original	Specialized	Measured	Estimated
add	12.2	4.6	2.7	$[2.7, 2.7]$
scanner	1.5	0.9	1.7	$[1.5, 4.1]$
polish-int	59.1	8.7	6.8	$[5.1, \infty]$

For the `add` program the speedup factor is *independent* of the dynamic input, and converges to 2.7 as the static input grows. Hence the very tight interval.

The interval for the '`scanner`' is quite satisfactory. If a specification of unambiguous tokens given, very little can be done at *mix*-time, and thus the speedup is close to the lower bound (as in the example). On the other hand, if the supplied table contains many "fail and backtrack" actions, the upper bound can be approached (not shown).

The upper bound for '`polish-int`' is correctly ∞ as the interpreter's code for handling unconditional jumps is completely static:

```
while (program[pp] != HALT)
   switch (program[pp])
      {
      case ...
      case JUMP:  pp = program[pp+1]; break;
      case ...
      }
```

Thus, an unbounded high speedup can be obtained by specialization with respect to a program with "sufficiently" many unconditional, consecutive jumps. To justify that the seemingly non-tight speedup interval computed by the analysis is indeed reasonable, we applied the "polish-form" interpreter to three different programs, *i.e.* three different static inputs. Each program exploits different parts of the interpreter.

The '`primes`' program computes the first n primes. The '`add`' program is the equivalent to the function '`add`' in Example 8.4. The '`jump`' program consists of a single loop with ten unconditional jumps. The measured speedups are as follows.

Example	Run-time		Speedup
	Original	Specialized	Measured
primes	59.1	8.7	6.8
add	51.5	5.5	9.2
jump	60.7	3.0	20.3

These experiments clearly demonstrate that the actual speedup *does* depend on the static input, as previously claimed.

8.3.5 Limitations and improvements

Even though the speedup analysis has demonstrated pragmatic success above, it has its limitations and suffers from some significant drawbacks. We have found that the lower bound computed by the analysis usually provides a fairly good estimate, but it is easy to construct examples which fool the analysis.

Loops are not related

Consider for example the program fragments below.

```
for (n = N; n; n--)          for (n = N; n; n--) S1;
   { S1; S2; }               for (n = N; n; n--) S2;
```

Suppose that S1 (static) and S2 (dynamic) do not interfere, meaning the two programs have the same observable effect. For the program to the left, the estimated speedup interval is $[4, 4]$ (counting 1 for all kinds of statements). The corresponding interval for the program to the right is $[3, \infty]$, where ∞ is due to the completely static loop. The latter result is still *safe* but certainly less tight — and useful — than the former.

The problem is that loops are considered in isolation, and the analysis therefore fails to recognize that the two loops iterate the same number of times.

Approximating the number of loop iterations

The analysis often errs conservatively but correctly, and reports infinite speedup due to static loops in the subject program. However, in many cases a loop bound is either present in the program, as in

```
for (n = 0; n < 100; n++) ...
```

or can be computed. This could be used to enhance the accuracy of the analysis since bounded static loops could be left out of the speedup computation, or added as the speedup, in the case of no residual loops.

Generalized constant folding analysis [Harrison 1977,Hendren *et al.* 1993] may be applied for approximation of a loop's iteration space. We have not, however, investigated this further.

Relating loops

As exemplified above, a major shortcoming in the analysis is that all loops are treated as being completely unrelated. Another blemish in the method is that all loops contribute equally to the final approximation of the speedup. For example, in a program with two loops, the one iterated 2 times, speedup 2, the other iterated 1000 times, speedup 10, the actual speedup is clearly close to 10. The speedup analysis would report the safe but loose speedup interval $[2, 10]$. Methods for approximating the number of loop iterations could alleviate the problem.

8.3.6 Variations of speedup analysis

So far we have assumed that a *global* speedup is computed by the analysis. However, several variations of the analysis may be useful.

When partial evaluation is applied to huge programs, it may be valuable to have a speedup estimate for each function, or a collection of functions. On the basis of a function speedup interval, specialization can be applied to functions that contributes with a significant speedup, and functions with a low speedup can be binding-time engineered or left out of consideration. Combined with a frequency or execution time analysis, *e.g.* as computed by the UNIX command '`prof`', the time spend on experiments with specialization could be lowered.

The analysis can easily be extended to support this. The only change needed is that only intra-procedural loops are collected, and call expressions are given appropriate costs.

Depending on the programs, the result of the analysis may be more coarse since recursion is left out of the speedup approximation. On the other hand, it can be avoided that completely static functions distort the result of the global analysis.

As a final example, we consider use of speedup estimates for determining feasibility of loop unrolling. Suppose given a loop

```
  for (n = 0; n < 1000; n++) S
```

where `S` is a dynamic statement. Clearly, it is undesirable to unroll the loop above due to code duplication. In general, it impossible to say *when* a static loop should be unrolled. A reasonable strategy seems to be that loops only shall be unrolled if the speedup of the body if beyond a certain lower limit. In particular, a loop with a speedup of 1 should never be unrolled.

8.4 Predicting speedups in generating extensions

In the previous section we developed an analysis that estimates a speedup before the value of static variables are available. An advantage is that the analysis time is independent of both the static and dynamic variables, but the analysis err conservatively since it must account for all static values. In this section we outline an approach where the speedup is computed *during* specialization. More concretely, we shall assume that specialization is accomplished via execution of generating extensions.

8.4.1 Accounting for static values

The aim is, given a program p and static input s, to compute a speedup valid for all dynamic input d. When the actual values of static variables are available the problem with unbounded static loops can be aboided. A drawback is that the execution time of the analysis depends on the specialization time, but in our experience specialization is normally so fast that actually generating a residual program to obtain a speedup estimate is no obstacle in practice.

Even though unbounded speedup becomes impossible — unbounded speedup manifests itself by non-termination — a precise speedup cannot be computed. The residual program may (dynamically) choose between branches with different speedups. We describe an approach where "profiling" information is inserted into generating extensions.

8.4.2 Speedup analysis in generating extensions

When a generating extension is run, the number of a times a static loop is iterated can be counted. Unrolling of a loop gives straight line code in the residual program with a speedup that will diminish when residual loops are iterated sufficient many times, so static loops can be left out of the speedup computation. Beware, static loops may be unrolled inside dynamic loops.

We extend a generating function with a counter for every dynamic loop: a count for number of static statements executed, and a count representing the number of residual statements generated. When specialization of a dynamic loop has completed, the counters can be employed to calculate the speedup of the (residual) loop.

Similar to the analysis of the previous section, the analysis result is given as the smallest interval that includes the speedups of all loops. We have not experimented with this analysis in practice but we expect it to give reasonable results.

8.5 Related work

It has for long been suspected that partial evaluation at must can accomplish linear speedup, but it has apparently never been proved nor published before. Yuri Gurevich has, however, independently come up with a similar reasoning. To our knowledge this is a first attempt at automatic estimation of speedup in partial evaluation.

8.5.1 Speedup in partial evaluation

Amtoft [Hansen 1991] proves in the setting of logic programming that fold/unfold transformations at most can give rise to linear speedup. The same restrictions as imposed upon '`mix`' are assumed. Note, however, that unification is daringly assumed to run in constant time, which may not be completely in accordance with reality.

As proved in this chapter, partial evaluation can only give constant speedups, and is thus uninteresting from a classical complexity theory point of view.[9] Recently, Jones has shown that constant factors actually do add computation power to small imperative languages [Jones 1993]. It remain to be investigated whether the constant speedups encountered by program transformation can be employed to classify the strength of transformers. The answer appears to be negative since partial evaluation extended with *positive context propagation* strictly increases the power of possible transformations, but positive context propagation is believed to give linear speedups.

8.5.2 Speedup analysis versus complexity analysis

Automatic complexity analysis has received some attention during the last years. The aim is: given a program p and possibly some "size" descriptions of the input, to compute a *worst-case* complexity function $\mathcal{O}_p(\cdot)$ in terms of the input size n.

It is tempting to apply techniques from automatic complexity analysis to speedup estimation. However, as the following example illustrate, the linear speedup in partial evaluation does not fit well into ordinary complexity analysis. Consider a loop

```
for (; n; n--)  {  S1; S2; ... Sj; }
```

Assume the relative speedup obtained by specialization of the statement sequence `S1;` ...`Sj;` to be k. Then the relative speedup of the whole program will be approximately k regardless of the loop being static or dynamic. However, if the loop is static, complexity analysis of the residual program will produce the answer $\mathcal{O}(1)$, since the loop has been

[9]This should not be taken as constants do not matter in practice!

unrolled. If the loop is dynamic, the result will be $\mathcal{O}(n)$. Not much insight (about speedup, that is) is gained this way.

It is, however, most likely that the techniques from automatic complexity analysis can be adapted to aid the problem of speedup analysis.

8.5.3 Properties of optimized programs

Malmkjær has developed an analysis that can predict the *form* of residual programs [Malmkjær 1992]. The output of the analysis is a grammar which indicates the structure of specialized programs. The analysis has nothing to say about speedups.

Inlining or unfolding is part of both compilation and partial evaluation, and its effect on execution times has been studied. Davidson and Holler have applied a C function inliner to a number of test programs and measured its effect on execution times [Davidson and Holler 1988,Davidson and Holler 1992]. Notice that this is pure empirical results; no prospective savings due to inlining are estimated. Ball describes an analysis that *estimates* the effect of function inlining [Ball 1979]. The analysis relies, however, on call statistics, and is therefore not feasible for programs that exhibit long execution times.

8.6 Future work

There are several possible directions for continuation of the preliminary results presented in this chapter. We believe that automatic speedup estimation is an emerging field which will become even more important when partial evaluation is used by non-experts.

8.6.1 Costs of instructions

In Section 8.3 we use a simple statement cost assigning 1 to every statement. It is trivial to refine the function to picture the cost of various expressions. However, an assignment based on high-level syntax is not accurate, and may be influenced by optimizations. Consider each in turn.

In C, the same operation can often be expressed in several different ways. Consider for an example '`n--`' versus '`n = n - 1`'. Seemingly, it is reasonably to assign a larger total cost to the latter expression than the former. On the other hand, must compilers will generate as efficient machine code for the latter as the former, so in practice the cost of the expressions is comparable.

It is well-known that specialization of a program may enable optimization which otherwise were not possible, *e.g.* unrolling of a loop resulting in a large basic blocks. However, optimizations may have unexpected effects on speedups. Consider the loop below,

```
for (n = 0; n <= N; n++)
   { S1; S2 }
```

where specialization will result in the straight line code '$\texttt{S1}_0\texttt{;S2}_0\texttt{;}\ldots\texttt{;S1}_N\texttt{;S2}_N$'.

Suppose that '`S1`' is loop invariant. In the original program, it will then be moved outside the loop, and executed only once. However, in the specialized program, it will be executed N times. Apparently, partial evaluation *degrades* efficiency.[10]

We leave it to future work to investigate the relation between speedups and optimizations of both subject and residual programs. Some preliminary study is undertaken in Chapter 9.

8.6.2 Estimation of code size

Run times are often used as the key measurement for judging the success of automatic transformations and optimizations. However, in the case of realistic programs, the *code size* matters. Very large programs may result in unexpected loss of performance due to register allocation pressure, an increased number of cache fails, or considerable longer compilation times.

The size of a residual program is broadly speaking determined by two factors: the unrolling of static loops, and inlining (or unfolding) of functions and statements.[11] Intuitively, a linear relation between code size and speedup seems reasonably: the gain in efficiency has a price; but super-linear code size blowup should be prevented.

For many programs, however, there is no obvious relation between code size and speedup, making automation of generalization with respect to code impossible. A class of programs, the so-called *oblivious algorithms* [Jones *et al.* 1993, Chapter 13] exhibits good behaviour: they contain no test on dynamic data, and hence the code size grows linearly with the static input. However, there seems no obvious way to transform a non-oblivious program into a more "well-behaved" program.

8.6.3 Unsafe optimizations and super-linear speedup

The Speedup Theorem in Section 8.2 assumes that '`mix`' only performs safe reductions. For example, partial evaluation is not allowed to throw away possibly non-terminating expressions. However, in some cases it can be recognized, either automatically or manually, that it is safe to discard an expression, or perform other "unsafe" reductions. Moreover, it seems natural to let a partial evaluator share computation in a residual program when it can detect it safe to do so. The causes and effects between unsafe reductions and super-linear speedup remain to be investigated.

8.7 Conclusion

We have investigated *speedup* in partial evaluation. It was proved that if *mix* is based on the techniques: function specialization, function unfolding/transition compression and constant folding, *super-linear* speedup is impossible.

[10]See also Chapter 9.

[11]An example of the latter: specialization of a dynamic if causes the following statements to be inlined into the branches.

A simple, but useful *speedup analysis* has been developed and implemented. The analysis computes a speedup *interval* by estimating the relative speedup of simple loops, given a binding-time annotated program. It was shown that an interval is a safe approximation of the actual speedup. Further, we outlined how speedup estimation can be accomplished during execution of generating extension, yielding in tighter speedup intervals. We described some experiments which showed reasonable results, but we also pointed out that the analysis can fail miserably on some programs. We believe a speedup estimate is a valuable information for users of a partial evaluator systems, and hence, further investigation of this field should be undertaken.

Chapter 9

Partial Evaluation in Practice

The aim of program specialization is *efficiency.* An application of a partial evaluator is successful if the residual program is faster than the subject program. Furthermore, the system itself should preferably be efficient.

In the previous chapters we have seen several examples where partial evaluation apparently pays off. In practice, however, many unexpected aspects may influence the actual speedup. There is only one way to assess the usefulness of the technique: by applying it to realistic examples. We have implemented a prototype version of the system developed in this thesis. The system implements most of the described techniques, and has generated the examples reproduced in this chapter.

Like traditional optimization techniques, partial evaluation must in the average produce more efficient programs. In particular, residual programs less efficient than subject programs should never be generated. It is normally believed that specialization enables greater classical optimization, often substantiated by the large basic blocks partial evaluation tends to generate. Thus, even when specialization in itself does not give significant speedups, enabled optimizations may contribute to give an overall good result. We investigate the relation between specialization and optimization, and find some astonishing connections. In particular, we observe that partial evaluation may both disable some optimizations and degrade the performance of programs. This leads to a discussion about the order in which program optimizations should be applied.

It is well-known that partial evaluators are sensitive to changes in subject programs. A slight modification may make the difference between good and insignificant speedups. Furthermore, some programs are more amenable for specialization than others. We provide an example where specialization of two similar algorithms gives very different results.

This chapter is organized as follows. In Chapter 9.1 we provide an overview of the implementation and its status. Chapter 9.2 is concerned with the relationship between size and speedup from a practical point of view. In chapter 9.3 we describe some intriguing interferences between specialization and optimization. Several experiments are reported in Chapter 9.4. Finally, Chapter 9.5 holds the conclusion and lists topics for further work.

9.1 C-Mix: a partial evaluator for C

We have made a prototype implementation of the partial evaluator *C-Mix* described in this thesis. It is implemented in C++, and generates programs in C. Currently, the implementation consists of approximately 20,000 lines of code.

9.1.1 Overview

An overview of the system is provided by Figure 4. A parser builds an abstract syntax tree representing the subject program. During the parsing, type checking and type annotation are performed, and various attributes are initialized. Further, types are separated and a value-flow analysis is carried out.

Several program analyses are applied in succession: static-call graph analysis (Chapter 2), pointer analysis (Chapter 4), side-effect analysis (Chapter 6) and binding-time analysis (Chapter 5). The speedup analysis (Chapter 8) has been implemented in a previous version of *C-Mix*, and the in-use analysis has not been implemented.[1] The generating extension generator (Chapter 3) converts the binding-time annotated program into a C++ program. An annotated version of the subject program can be inspected in a separate window.

At the time of writing we have not implemented the separate binding-time analysis nor the incremental constraint-solver developed in Chapter 7.

9.1.2 C-Mix in practice

A typical session with *C-Mix* is shown in Figure 66. We specialize the binary search function which is used as an example in the following section. At the time of writing we have implemented a coarse beautifier which converts ugly residual programs into less ugly residual programs. The specialized program in this chapter are listed *as* generated by *C-Mix*. Notice that "beautifying" does not improve efficiency, only readability!

9.2 Speed versus size

The underlying principle in partial evaluation is specialization of program points to values. As a consequence, algorithms that are *dependent* on static input data but *independent* of dynamic input data specialize "well".

9.2.1 Data-dependent algorithms

Intuitively, partial evaluation yields good speedups on programs where many control-flow branches can be determined statically, and some function calls can be replaced by constants. Programs containing many control-flow decisions dependent on dynamic data

[1]The figure is not telling the whole truth!

Generation of generating extension:
```
$ cmix -bDD -m bsearch -o bsearch-gen.cc bsearch.c
```
An annotated version of the program can be inspected in an X11 window.
Compiling and linking:
```
$ g++ bsearch-gen.cc main-gen.cc -lcmix -o bsearch-gen
```
The ' `main-gen.cc` *' file contains a '* `main()` *'*
function that calls the generating extension with the static input.
The object files are linked with the C-Mix library.
Specialization:
```
$ ./bsearch-gen > bsearch-spec.c
```
The result is a C program.
Compilation and linking:
```
$ gcc bsearch-spec.c main.c -o bsearch-spec
```
The original ' `main()` *' function must be changed according to the changed type of '* `bsearch()` *'.*
Execution:
```
$ time ./bsearch-spec
23.796u 0.059s 0:24.88 97.0% 0+184k 0+0io 0pf+0w
```

Figure 66: A session with C-Mix

are less suitable for specialization since both branches of dynamic `if`'s are specialized, potentially leading to residual programs of exponential size.

Example 9.1 The tests of the loops in the matrix multiplication function below are data-independent of the matrices '`a`', '`b`' and '`c`'.

```
/* matrix_mult: multiply matrices a and b, result in c */
void matrix_mult(int m, int n, int *a, int *b, int *c)
{
   int i, j, k;
   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
         for (c[i*m+j] = k = 0; k < n; k++)
            c[i*m + j] += a[i*m + k] + b[k*m + j];
}
```

Specialization with respect to static '`m`' and '`n`' will unroll the loops and produce straight line code. **End of Example**

An algorithm where the control-flow does not depend on dynamic input is called *oblivious* [Jones *et al.* 1993, Chapter 13]. Specialization of an oblivious algorithm results in straight line code of size polynomial with respect to the input. The matrix multiplication in Example 9.1 is oblivious. Huge basic blocks usually enable classical local data-flow optimizations, *e.g.* elimination of common subexpression and partial redundancies [Aho *et al.* 1986], and may also give better exploitation of pipelined computers [Berlin and Weise 1990]. Due to the fast development of larger computers, code size is often given less priority than speed.[2] In the next section, however, we argue that huge basic blocks is not always a good thing.

[2]If your program can run; buy some more memory!

```
/* bsearch: return entry of key in table[1000] */
int bsearch(int key, int *table)
{
   int low = 0, high = 1000 - 1, mid;
   while (low <= high) {
      mid = (low + high) / 2;
      if (table[mid] < key)
         low = mid + 1;
      else if (table[mid] > key)
         high = mid - 1;
      else
         return mid;
   }
   return -1;
}
```

Figure 67: Binary search function: variant 1

Clearly, most programs are not oblivious (but are likely to contain oblivious parts). Specialization of non-oblivious algorithms may result in residual program of exponential size (compared to the subject programs). Suppose, for an example, a program contains calls to a function f in both branches of a dynamic `if`. In the residual program, two (specialized) versions of f appear will if the static parametes differ. The residual program is probably more efficient than the subject program, but the code size may render specialization infeasible.

Algorithms fulfilling that the set of values bound to static variables are independent of dynamic input are called *weakly oblivious* [Jones *et al.* 1993, Chapter 13]. A desirable property of weakly oblivious algorithms is that specialization to static input s will terminate, provided normal execution terminates on s and some dynamic input. The reason is that the dynamic input does not influence the values assigned to static variables. Specialization of non-oblivious programs may loop since all static values must be accounted for.

Example 9.2 The binary search function in Figure 67 is non-oblivious when '`table`' is classified dynamic, since the tests depends on '`table`'. **End of Example**

9.2.2 Case study: binary search

Let us consider specialization of binary search functions. Figure 67 shows the "classical" binary search function, and Figure 68 depicts a variant [Bentley 1984]. Ignore for now the *C-Mix* specifier '`residual`'. Specialization with respect to "no" static input may seem useless, but notice that the table size (1000) is "hard-coded" into the programs. Thus, some static "input" is presence in the functions.

A remark. The function '`bsearch2`' was presented by Bentley as an example of manual code tuning. By hand he derived the residual program we generate automatically [Bentley 1984].

```
/* bseach2: return entry of key in table[1000] */
int bsearch2(int key, int *table)
{
   int mid = 512;
#ifdef CMIX
   residual
#endif
   int left = -1;
   if (table[511] < key) left = 1000 - 512;
   while (mid != 1) {
      mid = mid / 2;
      if (table[left + mid] < key) left += mid;
   }
   if ((left + 1) >= 1000 || table[left+1] != key) return -1;
   else return left + 1;
}
```

Figure 68: Binary search function: variant 2

Specialization of the '`bsearch1()`' function to a dynamic key and table, yields a residual program with the following appearance.[3]

```
/* This program was generated automatically */
int binsearch_1 (int v1, int *(v2))
{
   if (((v2)[499]) < (v1)) {
      if (((v2)[749]) < (v1)) {
         if (((v2)[874]) < (v1)) {
            if (((v2)[937]) < (v1)) {
               if (((v2)[968]) < (v1)) {
                  if (((v2)[984]) < (v1)) {
                     if (((v2)[992]) < (v1)) {
                        if (((v2)[996]) < (v1)) {
                           if (((v2)[998]) < (v1)) {
                              if (((v2)[999]) < (v1)) {
                                 return -1;
                              } else {
                                 if (((v2)[999]) > (v1)) {
                                    return -1;
                                 } else {
                                    return 999;
                                 }
                              }
                           } else {
                              if (((v2)[998]) > (v1)) {
                                 if (((v2)[997]) < (v1)) {
                                    return -1;
                                 }
   ...
```

[3]Programs are shown as generated by *C-Mix*.

The loop has been unrolled and the dynamic `if` specialized. Even though faster, this program is not especially admirable: it is huge, more than 10,000 lines. Consider now the variant of binary search shown in Figure 68. Part of the residual program is shown below.

```
/* This program was generated automatically */
int bsearch_1 (int v1, int *(v2))
{
   if (((v2)[511]) < (v1)) {
      if (((v2)[744]) < (v1)) {
         if (((v2)[872]) < (v1)) {
            if (((v2)[936]) < (v1)) {
               if (((v2)[968]) < (v1)) {
                  if (((v2)[984]) < (v1)) {
                     if (((v2)[992]) < (v1)) {
                        if (((v2)[996]) < (v1)) {
                           if (((v2)[998]) < (v1)) {
                              if (((v2)[999]) < (v1)) {
                                 return -1;
                              } else {
                                 if ((0) || (((v2)[999]) != (v1))) {
                                    return -1;
                                 } else {
                                    return 999;
                                 }
                              }
                           } else {
                              if (((v2)[997]) < (v1)) {
                                 if ((0) || (((v2)[998]) != (v1))) {
                                    return -1;
                                 } else {
                                    return 998;
                                 }
...
```

The result is equally bad.

Let us *suspend* the variable '`left`'. In Figure 68 this is indicated via the *C-Mix* specifier '`residual`' which unconditionally forces a variable to be classified dynamic. The result is shown below.

```
/* This program was generated automatically */
int bsearch_1 (int v1, int *(v2))
{
   int v4;
   v4 = -1;
   if (((v2)[511]) < (v1)) {
      v4 = 488;
      goto cmix_label3;
   } else {
      goto cmix_label3;
   }
```

```
cmix_label3:
  if (((v2)[(v4) + (256)]) < (v1)) {
     v4 += 256;
     goto cmix_label6;
  } else {
     goto cmix_label6;
  }
cmix_label6:
  if (((v2)[(v4) + (128)]) < (v1)) {
     v4 += 128;
     goto cmix_label9;
  } else {
     goto cmix_label9;
  }
...
```

This program is much better! Its size is $O(\log(n))$ of the table size, whereas the two residual programs above exhibit size $O(n)$.

The table below reports speedups and code blowups. The upper part of the table shows the result of specialization with dynamic '`key`' and '`table`'. The runtimes are user seconds, and the sizes are number of program lines. All experiments were conducted on a Sun SparcStation II with 64 Mbytes of memory, and programs were compiled by the Gnu C compiler with option '`-O2`'.

Program	Runtime (sec)			Code size (lines)		
	Orig	Spec	Speedup	Orig	Spec	Blowup
Table dynamic						
`bsearch1()`	39.1	20.3	1.9	26	10013	385
`bsearch2()` (left stat)	40.4	23.8	1.6	28	10174	363
`bsearch2()` (left dyn)	40.4	26.5	1.5	28	98	3.5
Table static						
`bsearch1()`	39.1	13.5	2.8	26	10013	385
`bsearch2()`	40.4	10.7	3.7	28	10174	363

The classical binary search function (Figure 67) gave the largest speedup: 1.9 compared to 1.5. The price paid for the 6 seconds the specialized version of '`bsearch1`' is faster than the specialized version of '`bsearch2`' is high: the residual version of '`bsearch1()`' is 385 times larger; the residual version of '`bsearch2()`' is only 4 times larger.

The lower part of the table shows speedups by specialization to a static '`table`'. Variant 2 yields a slightly larger speedup, but the programs are comparable with respect to code size.

The pleasing result of this small experiment is the good speedup obtained by specialization of '`bsearch2()`' despite the modest code blowup. The negative result is that it by no means is obvious that the classical binary search function should be replaced with a (slightly) less efficient routine before specialization. Even though it was not hard to discover that '`left`' should be suspended by looking at an annotated version of the program, it is not obvious that such genius suspensions can be automated.

9.3 Specialization and optimization

Program specialization aims at making programs more efficient; not slower. In this section we study the interference between partial evaluation and classical optimizations such as loop invariant motion, partial redundancy elimination and common subexpression elimination. Traditional partial evaluation technology subsumes (inter-procedural) constant folding, (some) dead code elimination, and loop unrolling.

9.3.1 Enabling and disabling optimizations

First a small illuminating example.

Example 9.3 Consider specialization of the following function where both parameters are classified dynamic. The specialized function is shown in the middle. Since the loop is static it has been unrolled.

```
int foo(int x, int y)     int foo_1(int x, int y)     int foo_2(int x, int y)
{                         {                           {
   int i = 0;                                            x = 1;
   while (i < 2)             x = 1; y += 2               y += 2;
      { x = 1; y += 2; }     x = 1; y += 2;              y += 2;
   return x + y;             return x + y;               return x + y;
}                         }                           }
```

Observe that the expression '`x = 1`' is loop invariant. To the right, the residual program obtained by loop hoisting the invariant *before* specialization is shown. Apparently, function '`foo_2()`' is preferable to function '`foo_1()`'.

Suppose that '`x = 1`' was an "expensive" loop invariant. Then it might be the case that '`foo()`' is *more efficient* than '`foo_1()`', due to the repeated computations of the invariant. However, if we apply dead code elimination to '`foo_1()`', it will be revealed that the first '`x = 1`' is dead, and we end up with '`foo_2()`'. **End of Example**

Lesson learned: specialization techniques may both *enable* and *disable* optimizations.

Partial evaluation normally enables local optimizations (inside basic blocks) since it tends to produce large(r) basic blocks, *e.g.* due to unrolling of loops. Furthermore, larger basic blocks give better exploitation of pipelined and parallel machines. As seen above, it can also disable global (across basic blocks) optimizations (loop invariant code motion). Seemingly, the optimizations being disabled by specialization should be applied *before* partial evaluation. Can a classical optimization applied to the subject before partial evaluation even *improve* residual programs? The example above seems to indicate an affirmative answer.

Example 9.4 Binding-time analysis takes both branches of conditionals into consideration. Suppose that dead-code elimination removes a branch of an `if` (possibly due to the result of a constant folding analysis) containing an assignment of a dynamic value to a variable. This may enable a more static division. **End of Example**

9.3.2 Order of optimization

It is well-known that classical optimizations interact and give different results depending on the sequence they are being applied [Whitfield and Soffa 1990]. For example, dead code elimination does not enable constant propagation: elimination of code does not introduce new constants.[4] On the other hand, constant propagatation may enable dead code elimination: the test of an `if` may become a constant. In the Gnu C compiler, the jump optimization phase is run both before and after common subexpression elimination.

Let us consider some classical optimization techniques and their relation to partial evaluation. In the following we are only interested in the efficiency of the residual program. For example, applying common subexpression elimination (to static expressions) before specialization may reduce the specialization time, but does not enhance the quality of residual code.

Common Subexpression Elimination (CSE). Partial evaluation is invariant to common subexpression elimination (of dynamic expressions). Thus, CSE does not improve beyond partial evaluation. On the other hand, specialization may enable candidates for elimination due to increased size of basic blocks.

Invariant Code Motion (ICM). Since specialization tends to remove loops, new candidates for ICM are not introduced. On the contrary, as seen in Example 9.3, partial evaluation may disable ICM.

Constant Propagation (CP). Partial evaluation includes constant propagation; thus there is no reason to apply CP to subject programs. On the other hand, residual programs may benefit from CP (constant folding) due to "lifted" constants, and the fact that binding-time analysis is conservative.

Loop Fusion (LF). Loop fusion merges two loops, the objective being elimination of control instructions. Partial evaluation is invariant to LF (on dynamic loops), and henceforth LF before specialization does not improve beyond LF applied to residual programs.

Consider the matrix multiplication program in Example 9.1, and suppose the inner loop is suspended, *e.g.* to reduce code size. The residual program consists of a sequence of loops.

```
for (c[0] = k = 0; k < N; k++) c[0] += ...
for (c[1] = k = 0; k < N; k++) c[1] += ...
for (c[2] = k = 0; k < N; k++) c[2] += ...
```

Loop fusion can bring these together.

Loop Unrolling (LU). Partial evaluation includes loop unrolling. Due to constant folding, specialization may enable loop unrolling not present in the subject program. The same effect can be achieved by double specialization.

Strip Mining (SM). The aim of strip mining is to decrease the number of logically related instructions such that *e.g.* the body of a loop can be in the instruction cache, or the active part of a table can be held in registers. Partial evaluation seems to produce a negative effect. It generates large basic blocks and unrolls loops, decreasing the likelihood for cache hits. On the other hand, exploitation of pipelining may be enhanced.

[4]However, consider the interaction with constant folding!

Optimization	CSE	ICM	CP	LF	LU	SM	SR	RAA	LVA
PE before opt.	+	−	+	+	+	−	+	−	
PE after opt.		+					+/−		+

Figure 69: Interaction between partial evaluation and optimization

Strengh Reduction (SR). Strength reduction has been applied to the loop below.

```
for (i = 1; i < 6; i++)          for (i = 1; i < 6; i++)
   { n = 5 * i; ...}               { n = n + 5; ... }
```

Suppose that 'n' is dynamic. The following residual programs result from specialization.

```
n = 5; ...           n = n + 5; ...
n = 10; ...          n = n + 10; ...
n = 15; ...          n = n + 15; ...
n = 20; ...          n = n + 20; ...
n = 25; ...          n = n + 25; ...
```

Obviously, SR degrades the effect of partial evaluation in this case. However, suppose now that the loop has the following appearance.

```
for (i = j = 1; i < 6; i++, j++)   for (i = j = 1; i < 6; i++, j++)
   { n = 5 * j; ... }                 { n = n + 5; ... }
```

where 'n' and 'i' are static, and 'j' dynamic. In this case SR will be of benefit: 'n' is separated from the dynamic 'j' (without SR the variable 'n' would have to be classified dynamic).

Register Allocation/Assignment (RAA). The aim is to keep active variables in registers, to reduce the number of load/store instructions. Due to data structure splitting, the number of variables may grow, causing register allocation pressure, and an increased number of cache misses.

Live-Variable Analysis (LVA). We have seen that live-variable analysis improves the quality of residual code (with respect to code size).

The considerations are summarized in Figure 69. A '+' in the first line means that partial evaluation may enable some optimization otherwise not possible and therefore improve the efficiency of the code. A '−' means that partial evaluation potentially may disable some optimizations otherwise possible. A '+' in the second line means that specialization will benefit from the optimization. To our knowledge, optimization of programs before partial evaluation has not been exploited in any existing partial evaluator. Automated binding-time improvements can be seen as an exception, though.

As the case with strength reducion clearly shows, the decision whether to apply optimization before or after partial evaluation is non-trivial.

9.3.3 Some observations

As illustrated by strength reduction, optimizations may change binding time classification of variables. Another example of this is dead-code elimination: elimination of dead, dynamic code may allow a variable to be classified static. A natural procedure would be to (automatically) inspect binding times to detect expressions that may benefit from classical optimizations before specialization. However, this (may) require the binding-time analysis to be iterated, which may be expensive. We leave as future work to characterize the optimizations/transformations that profitably can be applied before binding-time analysis.

We suspect that transformation into a form where expressions are cleanly binding-time separated may enhance efficiency of residual programs. Two immediate problems pop up: the residual programs become less readable ("high level assembler"), and the provision of useful feedback is rendered hard.

9.4 Experiments

This section reports results obtained by experiments with *C-Mix*. We provide four examples: specialization of a lexical analyis, some numerical analysis routines, a ray tracer, and a system for ecological modelling.

9.4.1 Lexical analysis

Lexical analysis is often a time-consuming task in compilers, both for humans and the compiler. Carefully hand-coded scanners can out-perform automatically generated scanners, *e.g.* constructed via '`lex`', but in practice, lex-produced scanners are employed. The reason is straightforward: it is a tedious job to implement lexical analysis by hand. Waite has made an experiment where a hand-coded scanner was compared to a lex-generated scanner [Waite 1986]. He found that hand-coded scanner was 3 times faster.[5]

In this example we test whether specialization of a naive lexical analysis yields an efficient lexer.

We use the scanner developed by Pagan as example [Pagan 1990, Chapter 4]. It is a simple program '`Scan`' that uses a so-called 'trie' for representation of tokens. Conversion of a language description (*e.g.* in the same form as a lex-specification) into a trie is almost trivial. The drive loop consists of 5 lines of code.

The following experiment was conducted. We specified the C keywords (32 tokens) as a trie table and as a lex-specification. White space was accepted but no comments. The trie-table consisted of 141 entries, the lex scanner had 146 DFA states. The scanner was specialized with respect to the trie, giving the program '`Spec`'. The lex-generated scanner '`Lex`' was produced by '`flex`' using default setting (*i.e.* no table compression).

The results are shown below, and has been generated on a Sun SparcStation II with 64 Mbytes of memory, and programs were compiled by the Gnu C compiler with option '`-O2`'.

[5]Thanks to Robert Glück, who directed us to this paper.

Time					Size				
(sec)			Speedup		(lines)			Blowup	
Scan	Spec	Lex	Scan	Lex	Scan	Spec	Lex	Scan	Lex
9.3	6.0	11.6	1.6	1.9	239	3912	1175	16.3	3.3

As input we used 20 different keywords each appearing 40,000 times in an input stream. The times reported are user seconds. The speedups are between the specialized scanner, and '`Scan`' and '`Lex`', respectively. As can be seen, Pagan's scanner is comparable to the lex produced scanner. However, the specialized version is 1.6 times faster than the general version, and nearly twice as fast as the lex-produced scanner.

The sizes of the programs are determined as follows. The size of '`Scan`' includes the driver and the table. The size of '`Lex`' is the size of the file '`lex.yy.c`' (as output by '`flex`') and includes the tables and the driver. As can be seen, the specialized version of '`Scan`' is 3 times bigger than the lex-produced scanner.

9.4.2 Scientific computing

Scientific computing results in many programs that are prime candidates for optimization. In this example two algorithms for solving linear algebraic equations taken from a standard book and specialized. The experiment was carried out by Peter Holst Andersen from DIKU.

The program '`ludcmp`' implements LU decomposition of matrices. The program '`ludksb`' solves a system linear equations by forward and backward substitution. Both programs are "library" programs and stem from the book [Press *et al.* 1989].

Both programs were specialized using *C-Mix* with respect to matrix dimension 5. On a Sun SparcStation 10 with 64 Mbytes of memory, with programs compiled by the Gnu C compiler using option '`-O2`', the following results were measured.

Program	Time			Size		
	Orig	Spec	Speedup	Orig	Spec	Blowup
ludcmp	34.9	23.1	1.5	67	1863	27
ludksb	15.1	7.6	2.0	27	247	9

Specialization of the LU decomposition program gives a speedup of 1.5. The specialized version of the solving function is almost twice as fast as the original version. Most of the speedup is due to unrolling of loops. Since the LU decomposition algorithm has a complexity of $O(N^3)$, the size of the residual program grows fast. The speedup is, however, rather surprising given the simple algorithms.

9.4.3 Ray tracing

The ray tracer example has been developed by Peter Holst Andersen from DIKU, and was the first realistic application of *C-Mix* [Andersen 1993c].

A *ray tracer* is a program for showing 3-dimensional scenes on a screen. It works by tracing light rays from a view point through the view to the scene. The idea is to specialize with respect to a fixed scene. It should be noted that the ray tracer was highly optimized before partial evaluation was applied.

The experiments were performed on an HP 9000/735, and programs compiled by the Gnu C compiler with option '`-O2`'. Program sizes are given as the object file's size (Kbytes).

Scene	Time			Size		
	Orig	Spec	Speedup	Orig	Spec	Blowup
17	14.79	8.91	1.7	102.5	61.6	0.6
23	11.25	6.61	1.7	102.5	93.2	0.9
24	16.31	9.29	1.8	102.5	98.9	1.0
25	33.02	17.57	1.9	102.5	96.9	0.9
26	6.75	4.02	1.7	102.5	66.5	0.6
27	10.80	6.17	1.8	102.5	68.2	0.7
28	16.89	9.27	1.8	102.5	67.1	0.7

As can be seen, a speedup of almost 2 is obtained, that is, the specialized program is twice as fast as the original. The reduction in size is due to elimination of unused scenes in the original program.

9.4.4 Ecological modelling

The Ecological modeling system SESAME is a system for simulation of biological processes in the north sea. The system has been developed by the Netherlands Institute for Sea Research (NIOZ), and has kindly been made to our disposal in a collaboration between DIKU and NIOZ. The experiments reported have been conducted by Peter Holst Andersen from DIKU.

The system models the flow of gases and nutrients between fixed so-called boxes (areas). The simulation is performed by extrapolation of the development via the solution of a number of differential equations.

The ERSEM model (a particular ecological model) consists of 5779 lines of C code, and the SESAME libraries (*e.g.* a differential equation solver) consists of 6766 lines of code. The experiments were done on a Sun SparcStation 10 with 64 Mbytes of memory. The model simulated 10 days and 2 years, respectively.

Simulation time	Orig model	Spec model	Speedup
10 days	57,564	36,176	1.6
2 years	4,100,039	3,174,212	1.3

The speedup is modest, but still significant given the long execution times. Rather surprisingly, a substantial part of the speedup is due to specialization of the '`pow()`' function to a fixed base. Furthermore, inlining of functions turned out to be essential for good results. The sizes of the residual programs are comparable to the original model.

Specialization of a future version of SESAME including advanced similation features is expected to yield larger speedups.

9.5 Conclusion and further work

We have considered partial evaluation in practice, and reported on several experiments with *C-Mix*.

9.5.1 Future work

The implementation of the system can be improved in a number of ways. The separated analysis developed in Chapter 7 seems to be of major pragmatic value, and its implementation should be completed.

The investigation of the interaction between partial evaluation and traditional program optimization should be continued. More generally, the causes and effects between specialization and efficiency need a deeper understanding. We are considering various transformations of subject programs before both binding-time analysis and the generating-extension transformation to improve efficiency of residual programs.

The preliminary analysis of this chapter has cast some light on the deficiencies of the speedup analysis in Chapter 8, but further development is needed for accurate estimation of speedups.

The case study in Section 9.2 illustrated the dependency between partial evaluators and subject programs. Notice in particular that a variable was *suspended* — the aim of binding time improvements is normally the contrary. The example is discouraging with respect to further automation of binding-time improvements. When should static variables be suspended to avoid unacceptable code blowup? When and where should binding time improvements be applied? More practical experiments seem necessary in order to get a deeper understanding of what pays off.

9.5.2 Summary

We have provided a number of experiments that clearly demonstate the power of partial evaluation. The speedups reported here are considerably smaller than those obtained in the functional language community, but this is to be expected. A speedup of 2 is still, however, a significant improvement. The speedup should be compared with the improvements obtainable by classical optimizations, *e.g.* advanced register allocation typically shorten the execution time by less than 10 %. Finally, we would like to stress that we have considered realistic programs; none of the programs have been "cooked" to show good results.

The experiments have also shown that more development is needed to automate and simplify the application of specialization to realistic programs. The rewriting of code often necessary today should be reduced, and feedback indicating where specialization should be applied would be useful.

Chapter 10

Improvements and Driving

A partial evaluator is not a genius program transformer. Sometimes it gives good results, other times it fails to optimize even simple programs. Only rather naive transformations are performed, and in particular no decisions are based on intelligence or clever "insight". As well-known, partial evaluators are often sensitive to changes in subject programs. Experience has shown that minor modifications may propagate through a programs in most unexpected ways.

On the other hand, program specialization via partial evaluation is fully automatic unlike most stronger transformation paradigms. Limited speedups of 2, say, matter in practice. By nature, it is hard to obtain substantial speedups by optimization of low-level, efficient languages like C. Many of the "improvements" possible in functional languages originate from very general programming constructs, which are not present in C, *e.g.* pattern matching.

In this chapter we first consider *binding-time improvements* that aim to give better results. Gradually, we introduce more powerful improvements that require insight into the subject program, and therefore are less amenable for automation. Next we shift to consider the strictly stronger transformation technique *driving*.

The main contribution of this chapter is the initial steps toward automation of *driving* for C. The driving transformation technique is a part of supercompilation, that so far only have been automated to some extent. We show by a case example how *positive context propagation*, which is the essence of driving, can transform a naive string matcher into a Knuth, Morris and Pratt matcher — a task partial evaluation cannot accomplish.

Next we introduce *generating super-extensions* which is the supercompilation equivalent to generating extensions in partial evaluation. Some first developments are reported, and perspectives are discussed.

The aim of this chapter is *not* to present a catalogue of useful "tricks" and "hacks" that can be applied to subject programs. In our opinion it should not be necessary to rewrite programs in order to apply a program transformation. Ideally, it must be "strong" enough to detect "naive" program constructs, and handle these in a convenient way. Manual rewritings are not an option for a programmer trying to optimize big programs. Currently these goals are not fulfilled in the case of even very simple languages.

10.1 Introduction

As evident from Chapter 9, partial evaluation is no panacea. Specialization pays off in form of significant speedups on some programs, and in other cases partial evaluation achieves almost nothing. Theoretically, a program's run-time can be improved by a constant only, as proved in Chapter 8. Unfortunately, partial evaluation is rather sensitive to even minor changes of the input. Ideally a partial evaluator must itself bring programs into a form suitable for specialization.

In practice, binding-time improving rewriting must be applied on some programs to obtain good result. The aim of such rewriting is (normally) to reveal static information.

10.1.1 A catalogue of transformation techniques

Several program transformation techniques, stronger than partial evaluation, have been studied in the literature. The most have been formulated for small, purely functional or declarative languages, and not generalized to realistic applications. Furthermore, many of the methods seems less amenable for automation, and have only been demonstrated on contrived examples.

A list of program transformation paradigms are listed below, in order of "power".

1. *Classical optimizations*, for instance common subexpression elimination, loop hoisting and elimination of partial redundancies [Aho *et al.* 1986].

2. *Partial evaluation* based on known evaluation of expressions, reduction of unknown expressions, specialization and unfolding of functions [Jones *et al.* 1993].

3. *Driving* and *generalized partial evaluation* [Glück and Klimov 1993,Jones 1994].

4. *Supercompilation* [Turchin 1986].

5. *Fold/unfold* transformations [Burstall and Darlington 1977].

Notice that all the mentioned techniques are source-to-source optimizations; we do not here consider language-to-language transformations (compilation) that produce optimized low-level implementations of in-efficient high-level formalisms. A notable example of the latter is the APTS system by Paige and Cai, which compiles set-based formulas into efficient C [Cai and Paige 1993].

Even though partial evaluation includes many of the classical optimizations, it differs pronouncedly by the availability of partial input. Naturally, partial evaluation may exploit "static" input present in program in the form of tables or the like, but we will disregard this for a moment. On the other hand, the fold/unfold transformation methodology may produce considerably improvements without any program input, *e.g.* deforestation [Wadler 1988].[1]

[1]Deforestation aims at eliminating immediate data structures in functional languages, and can be formulated via a set of fold/unfold rules.

Driving can be seem as partial evaluation plus *context propagation*, which is closely related to *theorem proving.* The idea is that in the then-branch of an '`if (x < 22)`', we know that '`x`' must be less than 22 even though '`x`' is unknown. Thus, if the then-branch contains a test '`if (x == 22)`', theorem proving can be applied to show that the test must evaluate to false. Propagation of information into the then-branch is called *positive context propagation*, and into the else-branch *negative context propagation.* Clearly, it is harder to assert that "something" is false, so driving using only positive context propagation is not uncommon. Driving has been partly automated for (small) functional languages [Glück and Klimov 1993], but generalization to realistic languages remain, as well as a handle on the prevailing termination problem.

Driving can be seen as an instance of the general technique known as supercompilation. Currently, supercompilation has been formulated and implemented for the Refal language [Turchin 1986].

All the above transformation techniques can be formulated in the framework of fold/-unfold transformations.[2] Fold/unfold consists of four rules: instantiate, define, fold, and unfold. To our knowledge automation of the general technique has not been achieved. Some special cases, however, have been systematized. For example, a restricted version of deforestation has been incorporated in the Glasgow Haskell compiler [Gill *et al.* 1993].

10.1.2 An example of a genius transformation

An example of a genius — or "intelligent" — optimization in the framework of fold/unfold transformations is given in Figure 70.[3] We consider the general definition of the Fibonacci function

```
int fib(int n)
{
   if (n < 2) return n;
   else return fib(n-1) + fib(n-2);
}
```

For ease of presentation we assume a pair structure definition

```
struct Pair { int x, y; };
```

and a function '`p()`' that "pairs" two integers and returns a pair.

In step 1 we *define* a function '`iter()`'. This definition is a Eureka! step; it is invented because it seems useful for the further transformation. In step 2 we *instantiate* '`iter()`' to 2 and *unfold* the definition of '`fib()`'. In step 3, the function *definition* '`addx()`' eliminates the common subexpression '`fib(n-1)`'.

In step 4 we recognize the *iteration* in '`iter()`', and transform the function into a loop as step 5.[4] In step 6 we *unfold* '`iter()`' into '`fib()`'.

Finally, by unfolding of the sum function '`sum()`' and splitting of the pair, we end up with the iterative definition of '`fib`' in step 7.

[2]At least in the case of functional languages.

[3]This section is modelled after [Jones *et al.* 1993, Page 353].

[4]This is not a fold/unfold step, by an application of a well-known transformation schema.

```
1. int fib(int n)
   {
      if (n < 2) return n; else return sum(iter(n-1));
   }
   struct Pair iter(int n)                    int sum(struct Pair p)
   {                                          {
      return pair(fib(n),fib(n-1));              return p.x + p.y;
   }                                          }
2. struct Pair iter(int n)
   {
      if (n < 2) return pair(1,0);
      else return p(fib(n-1)+fib(n-2),fib(n-1));
   }
3. struct Pair iter(int n)                    struct Pair addx(struct Pair p)
   {                                          {  int t = p.x;
      if (n < 2) return pair(1,0);               p.x += p.y; p.y = t;
      else return addx(fib(n-1),fib(n-2));       return p;
   }                                          }
4. struct Pair iter(int n)
   {
      if (n < 2) return pair(1,0);
      else return addx(iter(n-1));
   }
5. struct Pair iter(int n)
   {
      int i; struct Pair x = pair(1,0);
      for (i = 2; i <= n; i++) x = addx(x);
      return x;
   }
6. int fib(int n)
   {
      if (n < 2) return n;
      else { int i; struct Pair x = pair(1,0);
             for (i = 2; i <= n - 1; i++) x = addx(x);
             return sum(x);
           }
   }
7. int fib(int n)
   {
      if (n < 2) return n;
      else {
         int i, x = 1, y = 0, t;
         for (i = 2; i <= n - 1; i++) t = x, x += y, y = t;
         return x + y;
       }
    }
```

Figure 70: Transformation of the Fibonacci function

The net effect is that an exponential algorithm has been transformed into a linear algorithm. This is far beyond what partial evaluation can accomplish, but the transformation also appears hard to automate.

10.1.3 Overview of chapter

This chapter is organized as follows. In Section 10.2 we present a case study: generation of KMP matchers by partial evaluation. In Section 10.3 we investigate driving, and automate the example from Section 10.2. Section 10.4 contains related work, and Section 10.5 discuss further work and conclude.

10.2 Case study: generation of a KMP matcher

We will study specialization of a naive pattern matcher to a fixed pattern. The matcher runs in time $O(mn)$, where m is the length of the pattern and n the length of the string. The specialized matcher runs in time $O(n)$. We did a similar experiment in Chapter 3, then we specialized '`strstr()`' with respect to a fixed '`needle`'. In this section we are slightly more ambitious: the goal is to automatically generate an efficient Knuth, Morris and Pratt (KMP) matcher [Knuth *et al.* 1977], which we did not achieve in Chapter 3.

To meet these goals we extend partial evaluation with *positive context propagation.* Positive context propagation means that the positive outcome of tests are propagated into the then-branch of `if`s. Via "theorem proving" this can be exploited to decide some otherwise dynamic tests. In this example it is used to shift the (static) pattern more than one step with respect to the (dynamic) string.

In this section we obtain the effect of positive context propagation by rewriting of the string matcher, *i.e.* we apply a *binding-time improvement.* In Section 10.3 we present some initial steps towards automation of the technique, also known as *driving.* Notice that partial evaluation (as described in Chapter 3) cannot achieve the effect of transforming a naive string matcher into a KMP matcher *unless* we apply some manual binding-time improvements. Thus, *driving* is a strictly stronger transformation technique.

10.2.1 A naive pattern matcher

We use a slightly modified version of the string matcher '`strindex`' presented in Kernighan and Ritchie [Kernighan and Ritchie 1988, Page 69]. In accordance with the '`strstr()`' standard library function, our version returns a pointer to the first match in the string, rather than an index. The function is listed in Figure 71. It is considerably simpler (and slower!) than the Gnu implementation we considered in Chapter 3, but it works the same way.

The matcher compares the characters of the pattern '`p`' and the string '`s`' one by one, and shift the pattern one step in case of a mis-match. Partial evaluation of '`strindex`' to static input '`p = "aab"`' yields the following residual program.

```
/* strindex: return pointer to t in s, NULL otherwise */
char *strindex(char *p, char *s)
{
   int k;
   for (; *s != '\0'; s++) {
      for (k = 0; p[k] != '\0' && p[k] == s[k]; k++);
      if (p[k] == '\0') return s;
   }
   return NULL;
}
```

Figure 71: A naive pattern matcher (Kernighan and Ritchie page 69)

```
char *strindex_0(char *s)
{
   for (; *s != '\0'; s++)
      if ('a' == s[0])
         if ('a' == s[1])
            if ('b' == s[2])
               return s;
   return 0;
}
```

The complexity of the result is $O(n)$ and the number of nested ifs is m.[5] Partial evaluation has unrolled the inner loop, which eliminates some tests and jumps. The residual program is more efficient than the original program, but the improvement is modest.

10.2.2 Positive context propagation

The problem of the naive pattern matcher is well-known: information is thrown away! The key insight, initially observed by Knuth, Morris and Pratt, is that when a mis-match occurs, the (static) pattern provides information about a prefix of the (dynamic) string. This can be employed to shift the pattern more than one step such that redundant tests are avoided.

Consider the inner `for` loop which rewritten into a if-goto becomes

```
   k = 0;
l: if (p[k] != '\0' && p[k] == s[k])
     {
        k++;
        goto l;
     }
   if ...
```

where the test is dynamic due to the dynamic string '`s`'. Hence, partial evaluation specializes the `if`.

[5]Specialization of this program depends crucially on algebraic reduction of the && operator.

```
/* strindex: return pointer to t in s, NULL otherwise */
char *strindex(char *p, char *s)
{
   char ss[strlen(p)+1], *sp;
   int k;
   for (; *s != '\0'; s++) {
      for (k = 0; p[k] != '\0' && (p[k] == ss[k] || p[k] == s[k]); k++)
         ss[k] = p[k];
      if (p[k] == '\0') return s;
      /* Shift prefix */
      for (sp = ss; *sp = *(sp + 1); sp++);
   }
   return NULL;
}
```

Figure 72: Naive pattern matcher with positive context propagation

Consider specialization of the then-branch. Even though 's' is dynamic we know that (at run-time) the content of 's[k]' must equal 'p[k]' — otherwise the test would not be fulfilled, and then the then-branch would not be taken. The idea is to *save* this information, and use it to determine forthcoming tests on 's[k]'. Gradually as the pattern is "matched" against the dynamic string, more information about 's' is inferred.

We introduce an array 'ss[strlen(p)]' to represent the known prefix of 's'. The array is initialized by '\0' (not shown). The modified program is shown in Figure 72.

The test in the inner loop is changed to use the (static) prefix 'ss' of 's' if information is available, and the (dynamic) string 's' otherwise.

```
l: if (p[k] != '\0' && (p[k] == ss[k] || p[k] == s[k]))
```

First, the current position of the pattern 'p[k]' is compared to 'ss[k]'. If the test is true, algebraic reduction of '||' will assure that the test comes out positively. If no information about the string 's' is available (which manifest itself in 'ss[k]' being 0), a dynamic test against the string 's[k]' is performed.

In the body of the loop, the prefix string 'ss[k]' is updated by 'p[k]', and if the match fails, the prefix is shifted (like the pattern is shifted with respect to the string).

The binding-time improvement amounts to *positive context propagation* since we exploit that 'p[k] == s[k]' in the then-branch of the if statement.

10.2.3 A KMP matcher

Specialization of the improved matcher to the pattern 'p = "aab"' gives the following residual program.[6]

[6]We have restructured the output and renamed variables to aid readability

```
char *strindex_1 (char *s)
{
   while (*s != '\0')
      if ((0) || ('a' == s[0]))
        cmix_label7:
         if ((0) || ('a' == s[1]))
            if ((0) || ('b' == s[2]))
               return v1;
            else {
               s++;
               if (*s != '\0')
                  goto cmix_label7;
               else
                  return 0;
            }
         else
           s++;
      else
         s++;
   return 0;
}
```

The matcher is almost perfect — the only beauty spot being the test immediately before the `goto cmix_label7`. To remove this, *negative context propagation* is required. The careful reader can recognize a KMP matcher. In the case of a match '`aa`' where the next character is not a '`b`', the match proceeds at the same position; not from the initial '`a`'.

Some benchmarks are shown below. All experiments were conducted on a Sun SparcStation II with 64 Mbytes of memory, and programs were compiled by the Gnu C compiler using option '`-O2`'. Each match was done 1,000,000 times.

Input		Runtime (sec)			
Pattern	String	Naive	Spec	KMP	Speedup
aab	aab	1.8	0.9	0.7	2.5
aab	aaaaaaaaab	10.2	5.3	4.0	2.5
aab	aaaaaaaaaa	12.5	6.5	4.9	2.5
abcabcacab	babcbabcabcaabcabcabcacabc	19.2	10.2	7.3	2.6

'`Naive`' is the naive matcher, '`Spec`' is the specialized version of '`Naive`', and '`KMP`' is the specialized version of the improved version of '`Naive`'. The speedup is between '`Naive`' and '`KMP`'.

The experiment is similar to the one carried out in Chapter 3. We see that the naive matcher actually is faster than the Gnu implementation of '`strstr()`'. The reason being the (expensive) calls to '`strchr`'. In practice, the Gnu version will compete with the naive version on large strings. However, compare the residual programs. The KMP matcher is twice as fast as the specialization of Gnu '`strstr()`'.

Remark: the last example is the one used in the original exposition of the KMP matcher [Knuth *et al.* 1977].

10.3 Towards generating super-extensions

In the previous section we achieved automatic generation of a KMP matcher by manual revision of a naive string matcher. The trick was to introduce positive context propagation. In this section we consider adding positive context propagation to partial evaluation such that specialization of a naive matcher gives an efficient KMP matcher. Partial evaluation and context propagation is known as *driving.*

We present the preliminary development in form of a case study. More work is needed to automate the technique, and we encounter several problems during the way.

10.3.1 Online decisions

Driving is intimately connected with online partial evaluation. During specialization, variables may change status from "unknown" to "known", *e.g.* due to positive context propagation of a test '`x == 1`', where '`x`' previously was unknown. Further, information such as "'`x`' is unknown but greater than 1" must be represented somehow.

Existing experimental drivers, most notably the one developed by Glück and Klimov [Glück and Klimov 1993], are based on symbolic evaluation, and constraints on variables are represented via lists. However, as we have argued previously, specialization of realistic languages is best achieved via execution of generating extensions.

We will therefore pursue another approach, which uses a mixture of direct execution and online specialization techniques. In particular, the idea is to transform programs into generating extensions that include context propagation. We call a generating extension that performs *context propagation* a *generating super-extension.*

In this first development we will mainly restrict ourselves to positive context propagation (the extension to a limited form of negative context propagation is easy, though), and only consider propagation of scalar values. Thus, specialization of pointers, structs *etc.* is as described in Chapter 3.[7] Furthermore, we will assume that subject programs are pre-transformed such that no sequence points occur in test expressions, *e.g.* the || operator is transformed into nested `if`s.

For ease of presentation we program in the C++ language, which allows user-defined overloading of operators.

10.3.2 Representation of unknown values

We assume a binding-time analysis that classifies as static all computations that do not depend on unknown input. The problem is the representation of dynamic integer values in the generating super-extension. During the execution, they may change status to "known" and represent a (static) value, and other times they may be "unknown", but possibly constrained.

In the generating super-extension we represent dynamic (integer) values with the following class.

[7]From now on we ignore these constructs.

```
enum BTime { Equal, Less, Greater, Unknown };
class Int
{
   BTime btag;           /* binding time */
   int val;              /* value (if known) */
   Code code;            /* piece of code */
 public:
   Int(void)             { btag = Equal; val = 0; }
   Int(int v)            { btag = Equal; val = v; }
   BTime btime(void)     { return btag; }
   BTime btime(BTime b)  { return btag = b; }
   int value(void)       { return val; }
   int value(int v)      { return val = v; }
   void syntax(Code c)   { code = c; }
}
```

An '`Int`' consists of a *binding-time tag* which indicates the status of the value. '`Equal`' means the value has a known value which equals '`val`. '`Less`' means the value is unknown, but is less than '`val`. '`Unknown`' means that the value is unknown. In the latter cases, the field '`code`' will contain some code representing the value, *e.g.* the representation of '`x + y`', where both '`x`' and '`y`' are unknown.

Example 10.1 Consider the naive string matcher in Figure 71. We wish to infer information about the unknown input '`s`'. The problem is that even the size of the array '`s`' points to is unknown. Thus, the amount of information to be stored is unknown. Drivers for functional languages use dynamic allocation for representation of inferred information. We shall take a simpler (and more restrictive) approach. We will assume that '`s`' is a pointer to a known array with unknown content.

We define the '`s`' in the generating super-extension as follows

```
Code strindex(char *p, Int *s)
{ /*  s is pointer to array of unknown chars */
   int k;
   ...
}
```

where the array is assumed "big" enough. **End of Example**

10.3.3 Theorem proving

We define the following equality operator on '`Int`'.

```
Int operator==(Int &v1, Int &v2)
{
   Int v;
   switch (v1.btime()) {
      case Equal:
         switch (v2.btime()) {
            case Equal: v.btime(Equal); v.value(v1.value() == v2.value()); break;
            case Less:  if (v1.value() > v2.value())
                           v.btime(Equal), v.value(0);
```

```
                        else
                           v.btime(Unknown); v.syntax(...);
                        break;
     ...
     return v;
   }
```

The idea is to check the status of the operands during evaluation of the (dynamic) == operator. If both operands happen to be known, the test can be carried out, and the result is known. If the second operand is unknown but known to be less than n, the test is definitely false if n is less than the value of the first operand. And so on. Notice that comparisons are implemented by the underlying == operator (on integers), hence there is no "symbolic" evaluation of concrete values.

In a similar style can other operators, <, > *etc.* be defined.

Example 10.2 Driving is known to produce more terminating programs than subject programs. The problem is already apparent in the definition above. In the second case, if '`v2`' was non-terminating but its value (anyway) known to be less than the value of '`v1`', the net effect would be that the operand was discarded. Thus, non-terminating computation may be thrown away. **End of Example**

The operator definitions implement a (limited) form of *theorem proving* which can be exploited to decide dynamic tests.

Example 10.3 Consider the following program fragment in the generating extension for '`strindex`'. Suppose that '`p[k]`' is static and '`s[k]`' dynamic, but has (currently) a known value.

```
     cmixIf(p[Int(k)] == s[Int(k)], ... , ...)
```

Even though the test must be classified dynamic (by the binding-time analysis), it can (in the generating super-extension) be decided (in this case), since '`s[k]`' has a known value. Thus, there is no need to specialize both branches.

(Remarks. Since '`k`' is an integer value, it must be "lifted" to an '`Int`' in the array index expressions. We assume an operator '`[]`' on '`Int`' has been defined.) **End of Example**

In practice, this can be implemented by the '`cmixIf()`' code generating function by as follows:

```
cmixIf(Int test, Label m, Label n)
{
   switch (test.btime()) {
      case Equal: if (test.value()) goto m;  /* True! */
                  else goto n;  /* False! */
      case Less: if (test.value()<0) goto n;  /* False! */
                 else ... specialize m,n ... /* Don't know! */
      case Unknown: ... specialize m,n ... /* Don't know! */
      ...
   }
}
```

where the binding time of the test value is checked to see if the test can be decided.

This is similar to algebraic reduction of operators, and deciding dynamic tests, as described in Chapter 3.

10.3.4 Context propagation

Context propagation amounts to exploiting the positive and negative outcome of tests in the branches of dynamic `if`. Observe the following. If an `if` has to be specialized, evaluation of the test expression (in the generating super-extension) returns an '`Int`' containing the syntactic representation of the expression.

We can use this as follows. Before the then-branch is specialized, we traverse the (representation of the) test expression and perform positive context propagation. More precisely, we apply a function '`cmixPosContext()`' to the code for the test.

```
void cmixPosContext(Code e)
{
   switch (e.type) {
      case x == v:  x.btime(Equal); x.value(v); break;
      case x < v:   x.btime(Less); x.value(v); break;
      case x != 0:  x.btime(Unknown);
      ...
   }
 }
```

The function updates the value of the '`Int`'-variable '`x`', exploiting that the test expression "must" evaluate to true (at run-time). Similarly, a function '`cmixNegContext()`' can be defined.

Example 10.4 We have side-stepped an important problem. When should the context propagation functions be applied? The problem is that both functions *update* the values of the variables in the generating super-extension. One possibility is shown below.

```
void cmixIf(Int test, Label m, Label n)
{
   /* Apply theorem proving to decide the test */
   ...
   /* No luck. Then specialize both branches */
   else_store = cmixCopyStore();
   cmixPosContext(e);      /* Positive context propagation */
   cmixPendinsert(m);      /* Record for specialization */
   cmixRestoreStore(else_store);
   cmixNegContext(e);      /* Negative context propagation */
   cmixPendinsert(n);      /* Record for specializion */
   return new_label;
}
```

The active store is copied before the positive context propagation is performed, to be used for negative context propagation. This is a rather expensive solution.[8]

[8]It works, though, on small examples like 'strindex'.

The situation is slightly better if only positive context propagation is performed. In that case a copy of the active store for the else-branch can be inserted into the pending list; the positive context propagation performed, and specialization proceed immediately in the then-branch. **End of Example**

Some final remarks. The approach outlined in this section relies on the assumption that '`Int`' values are *copied* at specialization points as static variables. The representation described cannot represent information such as "v is less that 10 and greater than 0". Obviously, the definition of '`Int`' can be extended to support lists of constraints, at the price of extra memory usage and a more complicate mechanism for copying of '`Int`' values.

10.3.5 Is this driving?

We have shown a simple form of positive and negative context propagation in generating extensions. However, is this driving? And how does the methods relate to the benefits of generating extensions, *e.g.* direct execution and direct representation of static values?

The techniques are sufficient for (hand-coding) of a generating super-extension of the naive string matcher,[9] and it seems clear that the transformation can be automated. Obviously, more is needed to handle more realistic examples.

The theorem proving part of this section is currently used in *C-Mix* in a limited form. For example, without the algebraic reductions of the || operator in the '`strindex`' function, the results from specialization would be useless. At the time of writing we have only experimented with positive context propagation on contrived examples. Generalization is not straightforward.

It can be argued whether "known" values are executed directly or interpreted. In this section we have somehow side-stepped the problem by using a language supporting overloading of operators. If we had to code the similar functions in C, we would begin writing a symbolic evaluator. A main point is, however, that we use a binding-time analysis classification in parallel with "online" driving. Static values are represented directly, and there is no interpretation overhead on the evaluation of those.

10.4 Related work

Various "tricks" for improvement of partial evaluation are well-known among users of partial evaluators [Jones *et al.* 1993, Chapter 12]. Some of these have been automated but program specialization is still to some extend a "handicraft". For example, the Similix systems incorporate a CPS transformation that gives better exploitation of static values [Bondorf 1992].

Section 10.2 was inspired by the work by Glück and Klimov on supercompilation [Glück and Klimov 1993]. Coen *et al.* uses theorem proving to improve specialization [Coen-Porisini *et al.* 1991]. These two approaches are based on symbolic evaluation of subject programs. Kemmere *et al.* employ theorem proving for symbolic validation of

[9]A strong reservation: recall that we have assume the size of 's' is known — which was not the case in the original example.

Pascal programs [Kemmerer and Eckmann 1985]. Ghezzi *et al.* uses symbolic evaluation with value constraints to simplify programs [Ghezzi *et al.* 1985].

Glück and Jørgensen have experimented with automatic generation of stronger specializers using "information-carrying" interpreters [Glück and Jørgensen 1994]. Even though interesting results have been created, the method suffers from the need of a self-interpreter for the subject language.

10.5 Future work and Conclusion

We have considered driving and illustrated the possible structure of a generating super-extension. Generating super-extension differs most notably from driving via symbolic evaluation by the use of a binding-time analysis to classify as static definitely known values.

10.5.1 Further work

The work on generating super-extensions have recently been undertaken, and much work remains to extend the technique to a larger part of C. We believe that driving should be limited to base type values only, to avoid problems with representations, termination and memory usage.

Experiments are needed to evaluate the benefit of driving compared to ordinary partial evaluation.

10.5.2 Conclusion

We have initiated the development of generating super-extension for the C programming language, an extension to partial evaluation. By the means of a case study we showed generation of a KMP string matcher from a naive matcher — a result partial evaluation cannot produce — and outlined an approach which automates the generation.

We plan to incorporate positive context propagation into the *C-Mix* system in the future.

Chapter 11

Conclusion

We have developed a partial evaluator *C-Mix* for the C programming language. In Chapter 3 we studied program specialization for the C language, and described a generating extension transformation. The Chapters 4, 5, 6, 7, and 8 developed several analyses for C. A pointer analysis approximates the set of objects a pointer may point to at runtime. A binding-time analysis computes the binding times of expressions. A side-effect analysis and an in-use analysis approximates side-effects and the set of objects used by a function, respectively. A separate and incremental binding-time analysis support modular languages. A speedup analysis estimates prospective speedups. We provided several experimental results, and discussed the interaction between partial evaluation and program optimization in Chapter 9. Finally, we considered the stronger transformation technique driving in Chapter 10.

A list of contributions this thesis contains can be found in Chapter 1.

11.1 Future work

This thesis has developed and documented a partial evaluation system for the Ansi C programming language. In each chapter we have listed a number of specific topics for further work. In this section we provide more general directions for future work.

11.1.1 Program specialization and transformation

The generating-extension transformation was developed to overcome problems with a previous version of *C-Mix* based on traditional specialization via symbolic evaluation. It has very successfully proved its usefulness in practice. However, there is room for improvement. We outline some more prevailing, and urgent, problems here.

Our long term goal is that no rewriting of subject programs are necessary to obtain good results. Ways to automatical binding time improvements should be studied and implemented. In particular, separation of data structures seem important.

Specialization of imperative languages is problematic due to the memory usage. The problem is tightly coupled with the notion of non-oblivious algorithms. Many dynamic tests (may) cause extreme memory usage. Way to improve upon this is needed.

The termination problem hinders practical usage of partial evaluation. Even though it often is rather trivial to rewrite a program to prevent infinite specialization, this should be automated. The methods studied for functional languages seems insufficient for imperative languages. A fruitful way to proceed may be by general constant propagation that can guarantee that a variable only is bound to a finite number of (static) values.

11.1.2 Pointer analysis

Pointer analysis is the most important analysis for the C language. Other analyses depend on its precision, efficiency and the amount of gathered information.

The analysis we have developed is theoretically coarse but seems to work fine in practice. Experiments with the accuracy should be performed: does improved precision pay off, and what is the price in terms of efficiency? An obvious step is to extend the analysis into a flow-sensitive analysis, to avoid the spurious propagation of unrelated point-to information. Furthermore, our analysis is exhaustive in the sense that it computes information for all variants of functions. Clearly, this may be time consuming, and a way to limit the static-call graph to "relevant" variants should be investigated.

11.1.3 Binding-time analysis

The binding-time analysis has turned out to be very successful in practice. It is extremely fast, and its accuracy seems acceptable.

Work on integration of binding-time directed transformations aiming at improving the effect of specialization should be undertaken.

11.1.4 Data-Flow analysis

We have developed to classical analyses for the C programming language by exploring point-to information. The efficiency of the analyses can be improved as well as their precision. The in-use analysis should be coupled with analysis such as "last" use, "used only once" to improve the memory usage in generating extensions.

11.1.5 Separate analysis

The separate binding-time analysis has at the time of writing not been implemented in the *C-Mix* system, but is expected to be a major improvement. Work on separate pointer analysis is needed to obtain truly separate analysis. Further, extension into polyvariant analysis should be considered. A possibility is to build an inter-modular call-graph.

11.1.6 Speedup analysis

The speedup analysis is rather coarse, and often gives the minor informative answer that an arbitrary high speedup can be obtained. We have found that the lower bound often is a good estimate of the actual speedup, but tighter approximation would be useful.

Further, the result of the analysis should be applied to suspend specialization of program fragments, that contributes with little speedup.

Residual code size analysis is still an un-exploited area.

11.1.7 Experiments

The current implementation of the *C-Mix* system can be improved in a number of ways. Long term projects are to incorporate better separation of binding times. Transformation into a simpler language seems advantageous, but since experiments have clearly demonstrated the usefulness of informative feedback, we are unwilling to give up the high-level representation until a reasonable compromise have been found.

The study of the interaction between partial evaluation and classical optimization should be continued. Application of optimizations before partial evaluation seems to be a new approach.

11.1.8 Driving

The investigation of driving and generating super-extensions has recently begun. Much work is needed to automate and generalize the principles. Currently, theorem proving has turned out to be useful and reasonable to incorporate into generating extensions. More experiments are needed to assess the benefit of positive context propagation.

11.2 Final remarks

To our knowledge, *C-Mix* is the first partial evaluator handling a full-scale, pragmatically oriented language. The techniques are now so developed and mature that realistic experiments in professional software engineering can begin. We believe that experiments will reveal several areas for future work.

The applications of *C-Mix* in several experiments have given promising results, and demonstrated that the basic principles work. We hope that the results of this thesis may be useful in future software engineering.

Bibliography

[Aho *et al.* 1974] A.V. Aho, J.E. Hopcoft, and J.D. Ullman, *The Design and Analysis of Computer Algorithm,* Addison Wesley, 1974.

[Aho *et al.* 1986] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison Wesley, 1986.

[Allen and Cocke 1976] F.E. Allen and J. Cocke, A Program Data Flow Analysis Procedure, *Communications of the ACM* Vol 19, No 3 (March 1976) 137–147.

[Ammerguellat 1992] Z. Ammerguellat, A Control-Flow Normalization Algorithm and Its Complexity, *IEEE Transactions on Software Engineering* Vol 18, No 3 (March 1992) 237–251.

[Andersen 1991] L.O. Andersen, *C Program Specialization,* Master's thesis, DIKU, University of Copenhagen, Denmark, December 1991. DIKU Student Project 91-12-17, 134 pages.

[Andersen 1992] L.O. Andersen, *C Program Specialization,* Technical Report 92/14, DIKU, University of Copenhagen, Denmark, May 1992. Revised version.

[Andersen 1993a] L.O. Andersen, Binding-Time Analysis and the Taming of C Pointers, in *Proc. of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93,* pages 47–58, ACM SIGPLAN, June 1993.

[Andersen 1993b] L.O. Andersen, *Partial Evaluation of C,* chapter 11 in "Partial Evaluation and Automatic Compiler Generation" by N.D. Jones, C.K. Gomard, P. Sestoft, pages 229–259, *C.A.R. Hoare Series,* Prentice-Hall, 1993.

[Andersen 1993c] P.H. Andersen, Partial Evaluation Applied to Ray Tracing, August 1993. Student report.

[Andersen and Gomard 1992] L.O. Andersen and C.K. Gomard, Speed-up Analysis in Partial Evaluation: Preliminary results, in *Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'92),* pages 1–7, June 1992. YALEU/DCS/RR-909, Yale University.

[Andersen and Mossin 1990] L.O. Andersen and C. Mossin, Binding Time Analysis via Type Inference, October 1990. DIKU Student Project 90-10-12, 100 pages. DIKU, University of Copenhagen.

[Baier *et al.* 1994] R. Baier, R. Zöchlin, and R. Glück, Partial Evaluation of Numeric Programs in Fortran, in *Proc. of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation,* 1994. (To appear).

[Baker 1977] B.S. Baker, An Algorithm for Structuring Flowgraphs, *Journal of the ACM* Vol 24, No 1 (January 1977) 98–120.

[Ball 1979] J.E. Ball, Predicting the Effects of Optimization on a Procedure Body, in *Conf. Record of the Sixth Annual ACM Symposium on Principles of Programming Languages,* pages 214–220, ACM, January 1979.

[Banning 1979] J.P. Banning, An Efficient way to find the side effects of procedure calls and the aliases of variables, in *Conf. Record of the Sixth Annual ACM Symposium on Principles of Programming Languages,* pages 29–41, ACM, January 1979.

[Barzdin 1988] G. Barzdin, Mixed Computation and Compiler Basis, in *Partial Evaluation and Mixed Computation,* edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pages 15–26, North-Holland, 1988.

[Beckman *et al.* 1976] L. Beckman *et al.,* A Partial Evaluator, and Its Use as a Programming Tool, *Artificial Intelligence* 7,4 (1976) 319–357.

[Bentley 1984] J. Bentley, Programming Pearls: Code Tuning, *Communications of the ACM* Vol 27, No 2 (February 1984) 91–96.

[Berlin and Weise 1990] A. Berlin and D. Weise, Compiling Scientific Code Using Partial Evaluation, *IEEE Computer* 23,12 (December 1990) 25–37.

[Birkedal and Welinder 1993] L. Birkedal and M. Welinder, *Partial Evaluation of Standard ML,* Master's thesis, DIKU, University of Copenhagen, August 1993.

[Blazy and Facon 1993] S. Blazy and P. Facon, Partial Evaluation for the Understanding of Fortran Programs, in *Proc. of SEKE'93 (Software Engineering and Knowledge Engineering),* pages 517–525, Juni 1993.

[Bondorf 1990] A. Bondorf, *Self-Applicable Partial Evaluation,* PhD thesis, DIKU, University of Copenhagen, 1990.

[Bondorf 1992] A. Bondorf, Improving binding times without explicit CPS-conversion, in *1992 ACM Conference on Lisp and Functional Languages. San Francisco, California,* pages 1–10, June 1992.

[Bondorf and Dussart 1994] A. Bondorf and D. Dussart, Handwriting Cogen for a CPS-Based Partial Evaluator, in *Proc. of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation,* 1994.

[Bondorf and Jørgensen 1993] A. Bondorf and J. Jørgensen, Efficient analyses for realistic off-line partial evaluation, *Journal of Functional Programming* 3,3 (July 1993) 315–346.

[Bourdoncle 1990] F. Bourdoncle, Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity, in *PLILP'90,* pages 307–323, Springer Verlag, 1990.

[Bulyonkov and Ershov 1988] M.A. Bulyonkov and A.P. Ershov, How Do Ad-Hoc Compiler Constructs Appear in Universal Mixed Computation Processes?, in *Partial Evaluation and Mixed Computation,* edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pages 65–81, North-Holland, 1988.

[Burke 1993] M. Burke, Interprocedural Optimization: Eliminating Unnecessary Recompilation, *ACM Transaction on Programming Languages and Systems* 15,3 (July 1993) 367–399.

[Burstall and Darlington 1977] R.M. Burstall and J. Darlington, A transformation system for developing recursive programs, *Journal of Association for Computing Machinery* 24,1 (January 1977) 44–67.

[Cai and Paige 1993] J. Cai and R. Paige, Towards Increased Productivity of Algorithm Implementation, in *Proc. of the First ACM SIGSOFT Symposium on the Foundation of Software Engineering,* edited by D. Notkin, pages 71–78, ACM, December 1993.

[Callahan *et al.* 1986] D. Callahan, K.D. Cooper, K. Kennedy, and L. Torczon, Interprocedural Constant Propagation, in *Proc. of the SIGPLAN'86 Symposium on Compiler Construction, (ACM SIGPLAN Notices, vol 21, no 7),* pages 152–161, ACM, June 1986.

[Callahan *et al.* 1990] D. Callahan, A. Carle, M.W. Hall, and K. Kennedy, Constructing the Procedure Call Multigraph, *IEEE Transactions on Software Engineering* Vol 16, No 4 (April 1990) 483–487.

[Chase *et al.* 1990] D.R. Chase, M. Wegman, and F.K. Zadeck, Analysis of Pointers and Structures, in *Proc. of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation,* pages 296–310, ACM, June 1990.

[Choi *et al.* 1993] J. Choi, M. Burke, and P. Carini, Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects, in *Conf. Record of 20th. Annual ACM Symposium on Principles of Programming Languages,* pages 232–245, ACM, January 1993.

[Chow and Rudmik 1982] A.L. Chow and A. Rudmik, The Design of a Data Flow Analyzer, in *Proc. of the SIGPLAN'82 Symposium on Cimpiler Construction,* pages 106–113, ACM, January 1982.

[Coen-Porisini *et al.* 1991] A. Coen-Porisini, F. De Paoli, C. Ghezzi, and D. Mandrioli, Software Specialization Via Symbolic Execution, *IEEE Transactions on Software Engineering* 17,9 (September 1991) 884–899.

[Commitee 1993] X3J11 Technical Commitee, Rationale for American National Standard for Information systems — Programming Language C, Available via anonymous FTP, 1993.

[Consel 1993a] C. Consel, Polyvariant Binding-Time Analysis for Applicative Languages, in *Proc. of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM'93,* pages 66–77, ACM SIGPLAN, June 1993.

[Consel 1993b] C. Consel, A Tour of Schism: A Partial Evaluation System for Higher-Order Applicative Languages, in *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipuation,* pages 145–154, ACM, June 1993.

[Consel and Danvy 1993] C. Consel and O. Danvy, Tutorial Notes on Partial Evaluation, in *In Proc. of Symposium on Principles of Programming Languages,* pages 492–501, ACM, January 1993.

[Consel and Jouvelot 1993] C. Consel and P. Jouvelot, Separate Polyvariant Binding-Time Analysis, 1993. Manuscript.

[Cooper and Kennedy 1988] K.D. Cooper and K. Kennedy, Interprocedural Side-Effect Analysis in Linear Time, in *Proc. of the SIGPAN'88 Conference on Programming Language Design and Implementation,* pages 57–66, ACM, June 1988.

[Cooper and Kennedy 1989] K. Cooper and K. Kennedy, Fast Interprocedural Alias Analysis, in *Conf. Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages,* pages 49–59, ACM, January 1989.

[Cooper *et al.* 1986a] K.D. Cooper, K. Kennedy, and L. Torczon, The Impact of Interprocedural Analysis and Optimization in the R^n Programming Environment, *ACM Transactions on Programming Languages and Systems* 8,4 (October 1986) 491–523.

[Cooper *et al.* 1986b] K.D. Cooper, K. Kennedy, L. Torczon, A. Weingarten, and M. Wolcott, Editing and Compiling Whole Programs, in *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment,* edited by P. Henderson, pages 92–101, ACM, January 1986.

[Cooper *et al.* 1993] K.D. Cooper, M.W. Hall, and K. Kennedy, A Methodology for Procedure Cloning, *Computer Languages* Vol 19, No 2 (April 1993) 105–117.

[Cousot and Cousot 1977] P. Cousot and R. Cousot, Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in *Proc. of 4th Annual ACM Symposium on Principles of Programming Languages,* pages 238–252, ACM, January 1977.

[Cytron and Gershbein 1993] R. Cytron and R. Gershbein, Efficient Accommodation of May-Alias Information in SSA Form, in *In proc. of ACM SIGPLAN'93 Conference on Programming Language Design and Implementation,* pages 36–45, ACM, June 1993.

[Davidson and Holler 1988] J.W. Davidson and A.M. Holler, A Study of a C Function Inliner, *SOFTWARE — Practice and Experience* Vol 18(8) (August 1988) 775–790.

[Davidson and Holler 1992] J.W. Davidson and A.M. Holler, Subprogram Inlining: A Study of Its Effects on Program Execution Time, *IEEE Transactions on Software Engineering* Vol 18, no 2 (February 1992) 89–102.

[De Niel *et al.* 1991] A. De Niel, E. Bevers, and K. De Vlaminck, Partial Evaluation of Polymorphically Typed Functional Languages: The Representation Problem, in *Analyse Statique en Programmation Équationnelle, Fonctionnelle, et Logique, Bordeaux, France, Octobre 1991. (Bigre, vol. 74),* edited by M. Billaud *et al.,* pages 90–97, IRISA, Rennes, France, 1991.

[Edelson 1992] D.R. Edelson, Smart Pointers: They're Smart, but They're Not Pointers, in *Proc. of USENIX Association C++ Technical Conference,* 1992.

[Emami 1993] M. Emami, *A Practical Interprocedural Alias Analysis for an Optimizing/Parallelizing Compiler,* Master's thesis, McGill University, Montreal, September 1993. (Draft).

[Emami *et al.* 1993] M. Emami, R. Ghiya, and L.J. Hendren, *Context-Sensitive Interprocedural Points-to Analysis in the presence of Function Pointers,* Technical Report ACAPS Technical Memo 54, McGill University, School of Computer Science, 3480 University St, Montreal, Canada, November 1993.

[Ershov 1977] A.P. Ershov, On the Partial Computation Principle, *Information Processing Letters* 6,2 (April 1977) 38–41.

[Ferrante *et al.* 1987] J. Ferrante, K. Ottenstein, and J. Warren, The Program Dependence graph and its use in optimization, *ACM Transactions on Programming Languages and Systems* 9(3) (July 1987) 319–349.

[Freeman *et al.* 1990] B.N. Freeman, J. Maloney, and A. Borning, An Incremental Constraint Solver, *Communications of the ACM* 33,1 (January 1990) 54–63.

[Futamura 1971] Y. Futamura, Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler, *Systems, Computers, Controls* 2,5 (1971) 45–50.

[Ghezzi *et al.* 1985] C. Ghezzi, D. Mandrioli, and A. Tecchio, Program Simplication via Symbolic Interpretation, in *Lecture Notes in Computer Science,* pages 116–128, Springer Verlag, 1985.

[Ghiya 1992] R. Ghiya, *Interprocedural Analysis in the Presence of Function Pointers,* Technical Report ACAPS Memo 62, McGILL University, School of Computer Science, 3480 University St, Montreal, Canada, December 1992.

[Gill *et al.* 1993] A. Gill, J. Launchbury, and S.L. Peyton Jones, A Short Cut to Deforestation, in *Proceeding of Conference on Functional Programming Languages and Computer Architecture,* pages 223–232, ACM, June 1993.

[Glück and Jørgensen 1994] R. Glück and J. Jørgensen, Generating Optimizing Specializers, in *IEEE Computer Society 1994 International Conference on Computer Languages,* IEEE Computer Society Press, May 1994. (To appear).

[Glück and Klimov 1993] R. Glück and A.V. Klimov, Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree, in *Proc. of 3rd Int. Workshop on Static Analysis (Lecture Notes in Computer Science 724),* edited by G. Filè P. Cousot, M. Falaschi and A. Rauzy, pages 112–123, Springer Verlag, 1993.

[Gomard 1990] C.K. Gomard, Partial Type Inference for Untyped Functional Programs, in *1990 ACM Conference on Lisp and Functional Programming, Nice, France,* pages 282–287, ACM, 1990.

[Gomard and Jones 1991a] C.K. Gomard and N.D. Jones, Compiler Generation by Partial Evaluation: a Case Study, *Structured Programming* 12 (1991) 123–144.

[Gomard and Jones 1991b] C.K. Gomard and N.D. Jones, A Partial Evaluator for the Untyped Lambda-Calculus, *Journal of Functional Programming* 1,1 (January 1991) 21–69.

[Graham and Wegman 1976] S.L. Graham and M. Wegman, A Fast and Usually Linear Algorithm for Global Data Flow Analysis, *Journal of the Association for Computing Machinery* Vol 23, No 1 (January 1976) 171–202.

[Gross and Steenkiste 1990] T. Gross and P. Steenkiste, Structured Dataflow Analysis for Arrays and its Use in an Optimizing Compiler, *Software — Practice and Experience* Vol 20(2) (February 1990) 133–155.

[Gurevich and Huggins 1992] Y. Gurevich and J.K. Huggins, The Semantics of the C Programming Language, in *CSL'92 (Computer Science Logic),* pages 274–308, Springer Verlag, 1992. Errata in CSL'94.

[Hall 1991] M.W. Hall, *Managing Interprocedural Optimization,* PhD thesis, Rice University, April 1991.

[Hansen 1991] T.A. Hansen, Properties of Unfolding-Based Meta-Level Systems, in *Proc. of ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (Sigplan Notices, vol. 26, no. 9,* pages 243–254, ACM, 1991.

[Haral 1985] D. Haral, A Linear Time Algorithm for Finding Dominators in Flow Graphs and Related Problems, in *Proc. of the seventeenth Annual ACM Symposium on Theory of Computing,* pages 185–194, ACM, May 1985.

[Harrison 1977] W.H. Harrison, Compiler Analysis of the Value Range for Variables, *IEEE Transactions on Software Engineering* Vol 3(3) (May 1977) 243–250.

[Harrison III and Ammarguellat 1992] W. Harrison III and Z. Ammarguellat, A Program's Eye View of Miprac, in *Proc. of 5th International Workshop on Languages and*

Compilers for Parallel Compution (Lecture Notes in Computer Science, 757), edited by A. Nicolau U. Banerjee, D. Gelernter and D. Padua, pages 512–537, Springer Verlag, August 1992.

[Hecht and Ullman 1975] M.S. Hecht and J.D. Ullman, A Simple Algorithm for Global Data Flow Analysis Problems, *SIAM Journal of Computing* Vol 4, No 4 (December 1975) 519–532.

[Heintze 1992] N. Heintze, *Set-Based Program Analysis,* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, October 1992. (Available as technical report CMU-CS-92-201).

[Hendren and Hummel 1992] L.J. Hendren and J. Hummel, Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Tranformation of Imperative Programs, in *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation,* pages 249–260, ACM, June 1992.

[Hendren *et al.* 1992] L. Hendren, C. Donawa, M. Emami, Justiani G.R. Gao, and B. Sridharan, *Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations,* Technical Report ACAPS Memo 46, McGILL University, School of Computer Science, 3480 University St, Montreal, Canada, June 1992.

[Hendren *et al.* 1993] L. Hendren, M. Emami, R. Ghiya, and C. Verbrugge, *A Practical Context-Sensitive Interprocedural Analysis Framework for C Compilers,* Technical Report ACAPS Memo 72, McGILL University, School of Computer Science, 3480 University St, Montreal, Canada, July 1993.

[Henglein 1991] F. Henglein, Efficient Type Inference for Higher-Order Binding-Time Analysis, in *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523),* edited by J. Hughes, pages 448–472, ACM, Springer Verlag, 1991.

[Henglein and Mossin 1994] F. Henglein and C. Mossin, Polymorphic Binding-Time Analysis, in *Proc. of 5th. European Symposium on Programming (Lectures Notes in Computer Science, vol 788),* edited by D. Sannella, pages 287–301, Springer Verlag, April 1994.

[Hennessy 1990] M. Hennessy, *The Semantics of Programming Languages: an elementary introduction using structural operational semantics,* John Wiley and Sons Ltd, 1990.

[Holst 1991] C.K. Holst, Finiteness Analysis, in *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523),* edited by J. Hughes, pages 473–495, ACM, Springer Verlag, 1991.

[Holst and Launchbury 1991] C.K. Holst and J. Launchbury, Handwriting Cogen to Avoid Problems with Static Typing, in *Draft Proceedings, Fourth Annual Glasgow*

Workshop on Functional Programming, Skye, Scotland, pages 210–218, Glasgow University, 1991.

[Hood *et al.* 1986] R. Hood, K. Kennedy, and H.A Müller, Efficient Recompilation of Module Interfaces in a software Development Environment, in *Proc. of the SIGPLAN'86 Symposium on Compiler Construction, (ACM SIGPLAN Notices, vol 21, no 7),* pages 180–189, ACM, June 1986.

[Horwitz *et al.* 1987] S. Horwitz, A. Demers, and T. Teitelbaum, An Efficient General Iterative Algorithm for dataflow Analysis, *Acta Informatica* 24 (1987) 679–694.

[Hughes 1998] J. Hughes, Backward Analysis of Functional Programs, in *Partial Evaluation and Mixed Computation,* edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pages 187–208, North-Holland, 1998.

[Hwu and Chang 1989] W.W. Hwu and P.P. Chang, Inline Function Expansion for Compiling C Programs, in *Proc. of the SIGPLAN'89 Conference on Programming Language Design and Implementation,* pages 246–255, June 1989.

[ISO 1990] *Programming Languages—C,* ISO/IEC 9899:1990 International Standard, 1990.

[Jones 1988] N.D. Jones, Automatic Program Specialization: A Re-Examination from Basic Principles, in *Partial Evaluation and Mixed Computation,* edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pages 225–282, North-Holland, 1988.

[Jones 1989] N.D. Jones, Computer Implementation and Application of Kleene's S-m-n and Recursion Theorems, in *Logic from Computer Science,* edited by Y.N. Moschovakis, pages 243–263, Springer Verlag, 1989.

[Jones 1990] N.D. Jones, Partial Evaluation, Self-Application and Types, in *Automata, Languages and Programming. 17th International Colloquium, Warwick, England. (Lecture Notes in Computer Science, vol. 443),* edited by M.S. Paterson, pages 639–659, Springer Verlag, 1990.

[Jones 1993] N.D. Jones, Constant Time Factors *Do* Matter, in *STOC'93. Symposium on Theory of Computing,* edited by Steven Homer, pages 602–611, ACM Press, 1993.

[Jones 1994] N.D. Jones, The Essence of Partial Evaluation and Driving, in *Logic, Language, and Computation (Lecture Notes of Computer Science),* edited by N.D. Jones and M. Sato, pages 206–224, Springer Verlag, April 1994.

[Jones and Nielson 1994] N.D. Jones and F. Nielson, Abstract Interpretation: a Semantics-Based Tool for Program Analysis, in *Handbook of Logic in Computer Science,* Oxford University Press, 1994. 121 pages. To appear.

[Jones *et al.* 1989] N.D. Jones, P. Sestoft, and H. Søndergaard, Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation, *Lisp and Symbolic Computation* 2,1 (1989) 9–50.

[Jones *et al.* 1993] N.D. Jones, C.K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation, C.A.R. Hoare Series,* Prentice-Hall, 1993. (ISBN 0-13-020249-5).

[Kam and Ullman 1976] J.B. Kam and J.D. Ullman, Global Data Flow Analysis and Iterative Algorithms, *Journal of the Association for Computing Machinery* Vol 23, No 1 (January 1976) 158–171.

[Kam and Ullman 1977] J.B. Kam and J.D. Ullman, Monotone Data Flow Analysis Frameworks, *Acta Informatica* 7 (1977) 305–317.

[Kemmerer and Eckmann 1985] R.A. Kemmerer and S.T Eckmann, UNISEX: a UNIx-based Symbolic EXecutor for Pascal, *SOFTWARE—Practice and Experience* Vol 15(5) (May 1985) 439–458.

[Kennedy 1975] K. Kennedy, Node Listing applied to data flow analysis, in *Conference record of 2nd ACM Symposium Principles of Programming Languages,* pages 10–21, ACM, January 1975.

[Kennedy 1976] K. Kennedy, A Comparison of Two Algorithms for Global Data Flow Analysis, *SIAM Journal of Computability* Vol 5, No 1 (March 1976) 158–180.

[Kernighan and Ritchie 1988] B.W. Kernighan and D.M. Ritchie, *The C programming language (Draft-Proposed ANSI C), Software Series,* Prentice-Hall, second edition edition, 1988.

[Kildall 1973] G.A. Kildall, A Unified Approach to Global Program Optimization, in *Conference Record of the First ACM Symposium on Principles of Programming Languages,* pages 194–206, ACM, January 1973.

[Klarlund and Schwartzbach 1993] N. Klarlund and M.I. Schwartzbach, Graph Types, in *Conf. Record of the 20th. Annual ACM Sumposium on Principles of Programming Languages,* pages 196–205, ACM, January 1993.

[Knuth *et al.* 1977] D.E. Knuth, J.H. Morris, and V.R. Pratt, Fast Pattern Matching in Strings, *SIAM Journal of Computation* Vol 6, No 2 (June 1977) 323–350.

[Lakhotia 1993] A. Lakhotia, Constructing call multigraphs using dependence graphs, in *Conf. Record of 20th. Annual ACM Symposium on Principles of Programming Languages,* pages 273–284, ACM, January 1993.

[Landi 1992a] W.A. Landi, *Interprocedural Aliasing in the presence of Pointers,* PhD thesis, Rutgers, the State University of New Jersey, January 1992.

[Landi 1992b] W. Landi, Undecidability of Static Analysis, *ACM Letters on Programming Languages and Systems* 1, 4 (December 1992) 323–337.

[Landi and Ryder 1991] W. Landi and B.G. Ryder, Pointer-induced Aliasing: A Problem Classification, in *Eightteenth Annual ACM Sumposium on Principles of Programming Languages,* pages 93–103, ACM, January 1991.

[Landi and Ryder 1992] W. Landi and B.G. Ryder, A Safe Algorithm for Interprocedural Pointer Aliasing, in *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation,* pages 235–248, ACM, June 1992.

[Larus and Hilfinger 1988] J.R. Larus and P.N. Hilfinger, Detecting Conflicts Between Structure Acesses, in *In proc. of the SIGPLAN'88 Conference on Programming Language Design and Implementation,* pages 21–34, ACM SIGPLAN, June 1988.

[Launchbury 1990] J. Launchbury, *Projection Factorisations in Partial Evaluation,* PhD thesis, Dep. of Computing Science, University of Glasgow, Glasgow G12 8QQ, 1990.

[Launchbury 1991] J. Launchbury, A Strongly-Typed Self-Applicable Partial Evaluator, in *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523),* edited by J. Hughes, pages 145–164, ACM, Springer Verlag, 1991.

[Lengauer and Tarjan 1979] T. Lengauer and R.E. Tarjan, A Fast Algorithm for Finding Dominators in a Flowgraph, *ACM Transactions on Programming Languages and Systems* Vol 1, No 1 (July 1979) 121–141.

[Leone and Lee 1993] M. Leone and P. Lee, *Deferred Compilation: The Automation of Run-Time Code Generation,* Technical Report CMU-CS-93-225, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, December 1993.

[Malmkjær 1992] K. Malmkjær, Predicting Properties of Residual Programs, in *Proc. of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation,* pages 8–13, Yale University, Department of Computer Science, June 1992. (Available as technical report YALEU/DCS/RR-909).

[Malmkjær 1993] K. Malmkjær, Towards Efficient Partial Evaluation, in *Proc. of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation,* pages 33–43, ACM, June 1993.

[Marlowe and Ryder 1990a] T.J. Marlowe and B.G. Ryder, An Efficient Hybrid Algorithm for Incremental Data Flow Analysis, in *Conf. Record of the Seventeenth ACM Symposium on Principles of Programming Languages,* pages 184–196, ACM, January 1990.

[Marlowe and Ryder 1990b] T.J. Marlowe and B.G. Ryder, Properties of data flow frameworks, *Acta Informatica* 28 (1990) 121–163.

[Mayer and Wolfe 1993] H.G. Mayer and M. Wolfe, InterProcedural Alias Analysis: Implementation and Emperical Results, *Software—Practise and Experience* 23(11) (Nobemver 1993) 1201–1233.

[Meyer 1991] U. Meyer, Techniques for Partial Evaluation of Imperative Languages, in *Proc. of ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991),* pages 94–105, ACM, 1991.

[Meyer 1992] U. Meyer, *Partielle Auswertung imperative Sprachen,* PhD thesis, Justus-Liebig-Universität Giessen, Arndtstrasse 2, W-6300 Giessen, August 1992. In German.

[Mogensen 1989] T.Æ. Mogensen, *Binding Time Aspects of Partial Evaluation,* PhD thesis, Dept. of Comp. Science, University of Copenhagen, Mar 1989.

[Neirynck 1988] A. Neirynck, *Static Analysis of Aliases and Side Effects in Higher Order Languages,* PhD thesis, Computer Science, Cornell University, Ithaca, NY 14853-7501, Feb. 1988.

[Nielson and Nielson 1988] H.R. Nielson and F. Nielson, Automatic Binding Time Analysis for a Typed λ-Calculus, *Science of Computer Programming* 10 (1988) 139–176.

[Nielson and Nielson 1992a] F. Nielson and H.R. Nielson, *Two-Level Functional Languages,* Cambridge Computer Science Text, 1992.

[Nielson and Nielson 1992b] H.R. Nielson and F. Nielson, *Semantics with Applications,* John Wiley & Sons, 1992. ISBN 0 471 92980 8.

[Nirkhe and Pugh 1992] V. Nirkhe and W. Pugh, Partial Evaluation and High-Level Imperative Programming Languages with Applications in Hard Real-Time Systems, in *Conf. Record of the Nineteenth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1992,* pages 269–280, ACM, 1992.

[Olsson and Whitehead 1989] R.A. Olsson and G.R. Whitehead, A Simple Technique for Automatic Recompilation in Modular Programming Languages, *SOFTWARE–Practise and Experience* 19(8) (1989) 757–773.

[Pagan 1990] F.G. Pagan, *Partial Computation and the Construction of Language Processors,* Prentice-Hall, 1990.

[Plotkin 1981] G. Plotkin, *A Structural Approach To Operational Semantics,* Technical Report DAIMI FN-19, Computer Science Department, AArhus University, Denmark, Ny Munkegade, DK 8000 Aarhus C, Denmark, 1981. Technical Report.

[Pohl and Edelson 1988] I. Pohl and D. Edelson, A to Z: C Language Shortcommings, *Computer Languages* Vol 13, No 2 (1988) 5–64.

[Pollock and Soffa 1989] L.L. Pollock and M.L. Soffa, An Incremental Version of Iterative Data Flow Analysis, *IEEE: Transactions on Software Engineering* 15,12 (December 1989) 1537–1549.

[Press *et al.* 1989] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C,* Cambridge University Press, 1st edition, 1989.

[Ramalingam and Reps 1994] G. Ramalingam and T. Reps, An Incremental Algorithm for Maintaining the Dominator Tree of a Reducible Flowgraph, in *Conf. Record of the 21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages,* pages 287–296, ACM, January 1994.

[Richardson and Ganapathi 1989] S. Richardson and M. Ganapathi, Interprocedural Optimization: Experimental Results, *Software — Practise and Experience* 19(2) (February 1989) 149–169.

[Ross 1986] G. Ross, Integral C — A Practical Environment for C Programming, in *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* edited by P. Henderson, pages 42–48, ACM, January 1986.

[Ruf and Weise 1991] E. Ruf and D. Weise, Using Types to Avoid Redundant Specialization, in *Proc. of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91,* pages 321–333, ACM, June 1991.

[Ruggieri and Murtagh 1988] C. Ruggieri and T.P. Murtagh, Lifetime Analysis of Dynamically Allocated Objects, in *Proc. of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages,* pages 285–293, ACM, January 1988.

[Ryder 1979] B.G. Ryder, Constructing the Call Graph of a Program, *IEEE Transactions on Software Engineering* Vol SE-5, No 3 (May 1979) 216–226.

[Ryder and Paull 1986] B.G. Ryder and M.C. Paull, Elimination Algorithms for Data Flow Analysis, *ACM Computing Surveys* Vol 18, No 3 (September 1986) 277–316.

[Ryder *et al.* 1988] B.G. Ryder, T.J. Marlowe, and M.C. Paull, Conditions for Incremental Iteration: Examples and Counterexamples, *Science of Computer Programming* 11 (1988) 1–15.

[Rytz and Gengler 1992] B. Rytz and M. Gengler, A Polyvariant Binding Time Analysis, in *Proc. of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation,* pages 21–28, Yale University, Department of Computer Science, June 1992. (Available as technical report YALEU/DCS/RR-909).

[Sagiv and Francez 1990] S. Sagiv and N. Francez, A Logic-Based Approach to Data Flow Analysis Problems, in *Programming Language Implementation and Logic Programming: International Workshop PLILP'90,* edited by P. Deransart and J. Maluszyński, pages 278–292, Springer Verlag, August 1990.

[Schildt 1993] H. Schildt, *The Annotated ANSI C Standard,* Osborne, 1993. ISBN 0-07-881952-0.

[Sestoft 1988] P. Sestoft, Automatic Call Unfolding in a Partial Evaluator, in *Partial Evaluation and Mixed Computation,* edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pages 485–506, North-Holland, 1988.

[Sharir and Pnueli 1981] M. Sharir and A. Pnueli, *Two Approaches to Interprocedural Data Flow Analysis,* chapter 7, pages 189–233, Englewood Cliffs, NJ, 1981.

[Stallman 1991] R.M. Stallman, *Using and Porting Gnu CC,* Free Software Foundation Inc, 675 Mass Ave, Cambridge, 1.39 edition, january 1991.

[Tarjan 1983] R.E. Tarjan, *Data Structures and Network Algorithms,* Society for Industrial and Applied Mathematics, 1983.

[Turchin 1979] V.F. Turchin, A Supercompiler system based on the Langauge Refal, *SIGPLAN Notices* 14(2) (February 1979) 46–54.

[Turchin 1986] V.F. Turchin, The Concept of a Supercompiler, *ACM TOPLAS* 8,3 (July 1986) 292–325.

[Wadler 1988] P. Wadler, Deforestation: Transforming programs to eliminate trees, in *European Symposium on Programming,* pages 344–358, Springer Verlag, March 1988.

[Waite 1986] W.M. Waite, The Cost of Lexical Analysis, *SOFTWARE — Practice and Experience* Vol 16(5) (May 1986) 473–48.

[Weihl 1980] W.E. Weihl, Interprocedural Data Flow Analysis in the Presence of Pointers, Proceure Variables, and Label Variables, in *Conf. Record of the seventh Annual ACM Symposium on Principles of Programming Languages,* pages 83–94, ACM, January 1980.

[Whitfield and Soffa 1990] D. Whitfield and M.L. Soffa, An Approach to Ordering Optimizing Transformations, in *Second ACM SIGPLAN Symposium on PPOPP (SIGPLAN Notices Vol 25, No 3),* pages 137–146, ACM, March 1990.

[Yi 1993] K. Yi, *Automatic Generation and Management of Program Analyses,* PhD thesis, Center for Supercomuting and Development, University of Illinois at Urabana-Champaign, 1993.

[Yi and Harrison 1992] K. Yi and W.L. Harrison, *Interprocedural Data Flow Analysis for Compile-time Memory Management,* Technical Report 1244, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, August 1992.

[Zadeck 1984] F.K. Zadeck, Incremental Data Flow Analysis in a Structured Editor, in *Proc. of the ACM Sigplan '84 Symposium on Compiler Construction,* pages 132–143, ACM, June 1984.

[Zima and Chapman 1991] H. Zima and B. Chapman, *Supercompilers for Vector and Parallel Computers, Frontier Series,* ACM Press, 1991.

Dansk sammenfatning

Programudvikling er problematisk. På den ene side ønskes generelle og strukturerede programmer, der er fleksible og overkommelige at vedligeholde. Prisen for modularitet er *effektivitet.* Som regel er et specialiseret program, udviklet til en bestemt applikation, væsentlig hurtigere end et generelt program. Udvikling af specialiserede programmer er tidskrævende, og behovet for nye programmer synes at overstige kapaciticen af moderne programudvikling. Nye metoder er nødvendige til at løse denne såkaldte *programkrise.*

Automatisk partiel evaluering er en program specialiseringsteknik som forener *generalitet* med *effektivitet.* Denne afhandling præsenterer og dokumenterer en partiel evaluator for programmeringssproget C.

Afhandlingen beskaeftigerer sig med programanalyser og -transformation, og indeholder følgende hovedresultater.

- Vi udvikler en generating-extension transformation. Formålet med transformationen er specialisering af programmer.
- Vi udvikler en pointer-analyse. Formålet med pointer-analyse er at approksimere programmers brug af pointere. Pointer-analyse er essentiel for analyse og transformation af sprog som C.
- Vi udvikler en effektiv bindingtidsanalyse. Formålet med bindingtidsanalyse er at bestemme, om evaluering af et udtryk kan ske på oversættelsetidspunkt, eller først på køretid.
- Vi udvikler forskellige programanalyser til C, hvor vi udnytter information beregnet af pointer-analysen.
- Vi udvikler metoder for separate analyse og specialisering af programmer. Realistiske programmer er opdelt i moduler, hvilket stiller nye krav til programanalyse.
- Vi udvikler en automatisk effektivitetsanalyse, der estimerer den sandsynlige gevinst ved specialisering, og vi beviser en øvre grænse for den opnåelige hastighedsforøgelse.
- Vi betragter programtransformationen driving, hvilket er en stærkere transformationsteknik end partiel evaluering.
- Vi fremlægger eksperimentielle resultater opnået ved brug af en implementation af systemet.